MAPPING PARALLEL GRAPH ALGORITHMS TO THROUGHPUT-ORIENTED ARCHITECTURES

A Dissertation
Presented to
The Academic Faculty

By

Adam McLaughlin

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2015

Copyright © 2015 by Adam McLaughlin

MAPPING PARALLEL GRAPH ALGORITHMS TO THROUGHPUT-ORIENTED ARCHITECTURES

Approved by:

Professor David A. Bader, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Mark Clements
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Aaron Lanterman
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: September 22, 2015

To every teacher and mentor who has inspired me to reach beyond my comfort zone, challenged me personally and intellectually, and shown me life from an alternative perspective.

ACKNOWLEDGMENTS

I would firstly like to thank my advisor, David Bader, for his guidance and advice during my time at Georgia Tech. In addition to technical skills he has taught me quite a bit about professionalism and interpersonal relations - skills that will be invaluable to me in my career.

Secondly, I would like to thank the remaining members of my committee, Sudhakar Yalamanchili, Aaron Lanterman, Richard Vuduc, and Mark Clements, for their keen obsevatons and patience in scheduling and observing my dissertation proposal and defense.

Thirdly, I would like to thank my colleagues at Georgia Tech for their insightful discussions on research and philosophy. Special thanks to Jason Riedy for reviewing drafts of my papers and presentations and providing helpful comments (sometimes even in sed form!). Thanks to everyone else in the HPC lab: Lluis Munguia, Oded Green, Zhaoming Yin, James Fairbanks, Eisha Nathan, Anita Zakrzewska, Xing Liu, Jee Choi, Marat Dukhan, Jiajia Li, Piyush Sao, Aftab Patel, Patrick Flick, Indranil Roy, and Daniel Henderson; and thanks to those of you who I'm still fortunate to interact with from my days in the CASL.

Fourthly, I would like to thank all of the wonderful friends and mentors I have been fortunate enough to interact with on summer internships. Thanks to Pat McCormick and Kei Davis at Los Alamos National Lab; to Indrani Paul, Bobbie Manne, Peter Bailey, Joseph Greathouse, and many others at AMD; to Duane Merrill, Michael Garland, Saurav Muralidharan, Jin Wang, Jared Hoberock, Steven Dalton, Albert Sidelnik, Bryan Catanzaro, Norm Rubin, Tom Hart, Shankar Govindaraju, and plenty of others at NVIDIA; and finally to Michael Theobald, Joe Gagliardo, Andrew Parker, Naseer Siddique, Keun Sup Shim, Jochen Spengler, Bill Vick, Stan Wang, J.P. Grossman, Nick Johnson, Ramya Rangan, Aditya Limaye, Vipul Vachharajani, Kevin Yuh, all of the SAs, and all of the other brilliant minds I was introduced to at D.E. Shaw Research.

Thanks to all of my other friends for keeping me sane throughout my time in graduate

school. Special shoutouts to Phil Wilson, Justin Olsen, Bryan Fishberg, Amanda Szorentini, Amanda Leslie, John Konefal, Craig Diaz-Albertini, Josh Adler, Alex Pena, Jose Romero, Ashley Edwards, Afshin Mobramaein, Himanshu Sahni, Shray Bansal, Mikhail Jacob, Daniel Kohlsdorf, Robert Pienta, Brian Goldfain, Chad Stolper, Evan Downing, and many, many others.

Last, but certainly not least, I would like to thank my parents for supporting me throughout my entire life, not once discouraging me from following my bliss.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iv

LIST OF TABLES ix

LIST OF FIGURES xi

SUMMARY xiii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 RELATED WORK 4
2.1 Graph Traversal 4
2.2 GPU Implementations of Graph Algorithms 7
2.2.1 Strongly Connected Components 8
2.2.2 Betweenness Centrality 9
2.2.3 Additional Algorithms 10
2.3 Limitations of Contemporary GPU Graph Analysis 11

CHAPTER 3 SCALABLE AND HIGH PERFORMANCE GPU BETWEENNESS CENTRALITY 14
3.1 Background 15
3.1.1 Definitions 15
3.1.2 Brandes's Algorithm 16
3.2 Prior GPU Implementations 18
3.2.1 Vertex and Edge Parallelism 18
3.2.2 GPU-FAN 19
3.3 Methodology 22
3.3.1 Work-efficient Approach 22
3.3.2 Hybrid Approach 26
3.3.3 Sampling 32
3.4 Results 35
3.4.1 Experimental Setup 35
3.4.2 Scaling 36
3.4.3 Benchmarks 38
3.4.4 Multi-GPU Experiments 39
3.5 Related Work 42
3.6 Conclusions 43

CHAPTER 4 STREAMING BETWEENNESS CENTRALITY ON THE GPU 44
4.1 Related Work 45
4.1.1 Definitions 45
4.1.2 Brandes's Algorithm 46
4.1.3 Parallel Implementations 47

4.1.4 Dynamic Approaches 50
4.2 Dynamic Betweenness Centrality on the GPU 57
4.2.1 Updating the Number of Shortest Paths 58
4.2.2 Updating the Dependency Accumulation 62
4.3 Experimental Setup 65
4.4 Experimental Results 66
4.5 Conclusions 71

CHAPTER 5 OPTIMIZING TIME AND ENERGY FOR GPU BETWEENNESS CENTRALITY 72
5.1 Background 73
5.1.1 GPU Computing 73
5.1.2 Betweenness Centrality 74
5.2 Methodology 75
5.2.1 Coarse-grained Parallelism 75
5.2.2 Fine-grained Parallelism 77
5.3 Experimental Setup 79
5.4 Experimental Results 80
5.4.1 Static Experiments 80
5.4.2 Dynamic Experiments 82
5.5 Conclusions and Future Work 87

CHAPTER 6 FAST EXECUTION OF SIMULTANEOUS BREADTH-FIRST SEARCHES ON SPARSE GRAPHS 88
6.1 Background 90
6.1.1 Terminology 90
6.1.2 The Multi-Search Abstraction 90
6.1.3 Multi-Search Algorithms 91
6.2 Related Work 95
6.2.1 All-Pairs Shortest Paths 95
6.2.2 Parallel Abstractions for Graph Analysis 97
6.3 Multi-Search Implementation 98
6.4 Experimental Setup 104
6.5 Experimental Results 106
6.5.1 All-Pairs Shortest Paths 106
6.5.2 Betweenness Centrality 108
6.6 Conclusions 111

CHAPTER 7 AN ENERGY-EFFICIENT ABSTRACTION FOR SIMULTANEOUS BREADTH-FIRST SEARCHES 113
7.1 Background 114
7.1.1 The Multi-Search Abstraction 114
7.1.2 Related Work 115
7.2 Methodology 116
7.2.1 Implementation 117

7.2.2 Thresholding . . . 118
7.3 Evaluation . . . 122
7.3.1 Experimental Setup . . . 122
7.3.2 Experimental Results . . . 124
7.4 Conclusion . . . 126

CHAPTER 8 PARALLEL METHODS FOR VERIFYING THE CONSISTENCY OF WEAKLY-ORDERED ARCHITECTURES . . . 128
8.1 Background . . . 131
8.1.1 Constraint Graph . . . 131
8.1.2 TSOtool Workflow . . . 132
8.1.3 Inferred Edge Insertions . . . 135
8.2 Sequential Methodology . . . 136
8.2.1 Initial Algorithm . . . 136
8.2.2 Virtual Processors and Reverse Vector Time Clocks . . . 137
8.3 Facilitating Parallelism . . . 140
8.4 Parallel Methodology . . . 143
8.4.1 OpenMP . . . 144
8.4.2 CUDA . . . 145
8.5 Results . . . 147
8.5.1 Experimental Setup . . . 147
8.5.2 Experimental Results . . . 147
8.6 Conclusions . . . 154

CHAPTER 9 CONCLUSION . . . 156

REFERENCES . . . 158

LIST OF TABLES

Table 1 Correlation of vertex and edge frontier sizes with execution time for three randomly selected roots of different types of graphs. The size of the vertex frontier correlates positively with execution time regardless of the root or structure of the graph. 27

Table 2 Graph datasets used for this study 34

Table 3 Performance of edge-parallel and sampling methods for various graphs. Results are in MTEPS (Millions of Traversed Edges per Second). 38

Table 4 Multi-node performance for various graphs. Results are in GTEPS (Billions of Traversed Edges per Second). 41

Table 5 Suite of benchmark graphs 65

Table 6 Comparison of Dynamic CPU and Dynamic GPU Algorithms 66

Table 7 Comparison of Node Parallel GPU Updates to GPU Recomputation . . . 68

Table 8 GPUs used for this study 79

Table 9 Graph datasets used for this study 80

Table 10 Energy-efficiency of static BC computations on the GPU for various classes of networks 81

Table 11 Comparison of dynamic BC computations on the CPU and GPU of the Kayla platform 82

Table 12 Comparison of static and dynamic BC computations on the GPU of the Kayla platform 83

Table 13 Graph datasets used for this study. Nodes and edges are displayed in millions. 105

Table 14 Benchmark results for solving the APSP Problem. 108

Table 15 Benchmark results for computing Betweenness Centrality. Times are in seconds. The fastest result for each graph is presented in bold. 110

Table 16 Average speedup of the cooperative approach over existing frameworks. . 111

Table 17 Graph datasets used for this study. 122

Table 18 Timings for various methods of graph traversal in seconds. 124

Table 19 Energy Consumption for various methods of graph traversal in Joules. . . 126

Table 20 Speedup over TSOtool for our sequential and parallel implementations of inferring edges 150

Table 21 Parallel Speedups over Algorithm 23 150

Table 22 Application speedup over TSOtool for our sequential and parallel implementations 151

Table 23 Metadata regarding the twelve largest test cases 153

LIST OF FIGURES

Figure 1 Example Betweenness Centrality scores for a small graph 16

Figure 2 Illustration of the distribution of threads to units of work. Top: Vertex-parallel. Middle: Edge-parallel. Bottom: Work-efficient. 20

Figure 3 Evolution of vertex frontiers (as a percentage of total vertices) for different classifications of graphs 28

Figure 4 Comparison of Work-Efficient, Hybrid, and Sampling methods 33

Figure 5 Scaling by problem size for three different types of graphs 37

Figure 6 Multi-GPU scaling by number of nodes for various graph structures. Each node contains three GPUs. 40

Figure 7 BC speedup relative to one thread block 49

Figure 8 Distribution of scenarios for the graphs used in this study 52

Figure 9 Decomposition of work to parallel compute units 56

Figure 10 Portion of the graph that is touched for each Case 2 scenario 70

Figure 11 Power consumed as a function of the number of thread blocks launched . 76

Figure 12 Decomposition of work to parallel compute units 78

Figure 13 Percentage of vertices touched by Case 2 scenarios 84

Figure 14 Percentage of vertices touched by Case 3 scenarios 84

Figure 15 Left: *preferentialAttachment* Middle: *kron_g500-logn19* Right: *smallworld* 86

Figure 16 Several thread decompositions for the multi-search abstraction 101

Figure 17 Impact of scaling vertices and edges on performance for Erdős-Rényi graphs 103

Figure 18 Comparison of multi-search traversal techniques for two classes of networks 107

Figure 19 Relative effect of the threshold parameter T on performance. 121

Figure 20 Design flow for memory consistency verification 133

Figure 21 Example of a Rule 6 inferred edge insertion 135

Figure 22 Example of Rule 7 inferred edge insertions 135

Figure 23 Splitting of a physical processor in sequentially consistent virtual processors 138

Figure 24 Performance results for various test sizes 148

Figure 25 Scatter plot of edges added and performance 152

SUMMARY

The stagnant performance of single core processors, increasing size of data sets, and variety of structure in information has made the domain of parallel and high-performance computing especially crucial. Graphics Processing Units (GPUs) have become an exciting alternative to traditional CPU architectures for applications in this domain. Although GPUs are designed for rendering graphics, research has found that the GPU architecture is well-suited to solving dense systems of linear equations. Subsequently, general purpose programming models for GPU computing, such as CUDA and OpenCL, were developed to bring the potential performance enhancements from GPUs to a broader audience.

Recently, computational scientists have discovered that other classes of algorithms once thought to be unreasonable GPU workloads can also map well to throughput-based architectures. In particular, algorithms that search and analyze unstructured, graph-based data can leverage the high memory bandwidth of GPU architectures to see up to an order of magnitude performance improvement over their CPU counterparts.

Network analysis, particularly with the large number of threads offered by contemporary GPUs, comes with a number of challenges. Memory access patterns of graph algorithms tend to be data dependent and as a result programmers spend months of time attempting to obtain even a fraction of the peak theoretical throughput of the processor. Even worse, these algorithms tend to be specialized to a specific architecture, problem size, or data set. Indeed, different graph structures benefit from different parallelization schedules, making a general approach to high-performance graph analysis tremendously difficult.

This thesis focuses on GPU graph analysis from the perspective that algorithms should be efficient on as many classes of graphs as possible, rather than being specialized to a specific class, such as social networks or road networks. Using betweenness centrality, a popular analytic used to find prominent vertices of a graph, as a motivating example,

we show how parallelism, distributed computing, hybrid and on-line algorithms, and dynamic algorithms can all contribute to substantial improvements in the performance and energy-efficiency of these computations. We further generalize this approach and provide an abstraction that can be applied to a whole class of graph algorithms that require many simultaneous breadth-first searches. Finally, to show that our findings can be applied in real-world scenarios, we apply these techniques to the problem of verifying that a multiprocessor complies with its memory consistency model.

CHAPTER 1

INTRODUCTION

Real world data sets tend to contain massive, unstructured pieces of information. The analyses required to parse through this information are complicated, yet still need to meet the demands of businesses and governments in a reasonable amount of time. Graph-based abstractions have long been used as a general representation of unstructured data sets, but algorithms that operate on graphs have only recently been ported to fast multi-core and accelerator-based architectures. Current solutions to large network analysis problems tend to use supercomputers and other types of distributed memory CPU systems. Such solutions are costly and difficult to scale to larger problem instances; hence, new algorithms, system software, and hardware solutions are in high demand.

Distributed systems are a costly solution to large scale graph analysis because communication time between nodes is significant and they have a high rate of power consumption. Unfortunately, the end of Moore's law and Dennard scaling has lead to minimal performance and energy-efficiency gains in CPU design in the last decade. This thesis builds upon existing work on graph analysis using the Graphics Processing Unit (GPU) as the focal point of computation. Currently, GPUs have a memory bandwidth that is up-to-order of magnitude greater than CPUs, allowing for significant performance gains on memory-bound network analysis routines. GPUs were originally developed as parallel processors for rendering graphics and are a commodity (and thus, inexpensive) piece of hardware found in today's desktop computers.

Computing graph analytics on GPUs comes with a number of significant challenges, however. Firstly, the current state of GPU software is immature and challenging to write. GPU code is tremendously low-level; developers must write their own kernels, manually manage scratchpad memories, and have a detailed knowledge of the underlying architecture to obtain even a fraction of the processor's peak performance. Furthermore, there are few

widely-used libraries. Existing code tends to be specialized to a particular problem, data set, or hardware platform. As a consequence, developers tend to write their own versions of kernels for subroutines that are taken for granted when developing on traditional CPU architectures. Secondly, graph algorithms typically have data-dependent memory access patterns and lots of complicated and error-prone pointer arithmetic, making compile-time analysis, branch predictors, and caches less useful. Thirdly, code reuse is tremendously difficult. GPU kernels typically takes months for a domain expert to tune and optimize. Changing these kernels in a subtle way can lead to opaque and drastically different performance characteristics. This is especially true in the context of graph analysis where the metadata stored within each node tends to change from one application to another.

This thesis is a unified attempt to alleviate these aforementioned problems, making the following contributions:

- A work-efficient algorithm for Betweenness Centrality that does especially well for high-diameter graphs [1].
- Hybrid approaches to Betweenness Centrality that are performance portable to graph structures commonly seen in real world applications [1].
- A distributed Betweenness Centrality algorithm that shows linear speedups on a cluster of 192 GPUs [1].
- A dynamic algorithm that can quickly update BC scores due to minor variations in the graph rather than recomputing the scores entirely [2].
- A power analysis of the above techniques for comparison of their energy-efficiency to that of prior art [3].
- A simple abstraction that generalizes these methods into a framework that can be used on an entire class of problems requiring simultaneous breadth-first searches [4].

- An application of these techniques to quickly find inconsistencies in real world multiprocessor design [5].

We first focus on betweenness centrality, due to it's exploding popularity in the context of social network analysis; however, one of the thematic elements of this thesis is that the algorithms developed provide fast performance across a broad spectrum of graph data sets: road networks, meshes, social networks, crawls of the Internet, and more. Our findings from betweenness centrality are then generalized into a framework containing an abstraction for a broader class of graph algorithms. Specifically, we improve our methods and generalize them to any graph algorithm that requires many simultaneous breadth-first searches: transitive closures, diameter samplings, reachability queries, etc. Although our initial benchmark for success is performance, we additionally take energy consumption into account, showing that our methods save energy compared to CPU systems and are portable to embedded GPU architectures such as NVIDIA's Kayla platform as well as the Keneland Initial Delivery System (KIDS), a distributed system of 120 nodes, each containing 3 GPUs. Finally, we show that this work applies to real-world problems faced in industry by vastly accelerating the time required to verify a multiprocessor's memory consistency model.

This work shows novel methods for accelerating network analysis algorithms on GPU architectures, but also conveys that its contributions are a small part of a much larger, open problem. Further scaling to larger problems and systems for even better performance allows for a more interactive experience for end users who have an unquenchable thirst for insight.

CHAPTER 2

RELATED WORK

The GPU's ability to concurrently process many fine-grained parallel tasks and its high memory bandwidth compared to that of traditional CPUs has lead to several GPU implementations of graph algorithms in recent literature. Initial work focused mainly on graph traversal techniques. Since depth-first search is inherently sequential [6], this work has focused on efficiently mapping Breadth-First Search (BFS) algorithms to the underlying architecture. Additionally, more complicated graph algorithms such as Single-Source Shortest Paths, Betweenness Centrality, and Strongly Connected Components (SCCs) have since been efficiently implemented on the GPU. There has also been some work on improving GPU programmer productivity through the creation of libraries consisting of commonly used primitives such as prefix scan, reduction, and sorting [7, 8]. This chapter discusses the current state of the art in fine-grained parallel graph algorithms on throughput processors and the limitations of existing GPU libraries with respect to network analysis.

2.1 Graph Traversal

Since graph traversal is a key primitive used for the construction of higher-level graph algorithms, it is unsurprising that it has received significant attention in the literature [9–11]. The first well-known GPU implementation of Breadth-First Search was designed by Harish and Narayanan in 2007 [12]. In addition to BFS, they presented algorithms for finding shortest paths. Their implementation of BFS uses the Compressed Sparse Row (CSR) format, a representation that is still widely in use today, to store the graph; however, their BFS kernel trivially assigns a logical (i.e. software) thread to each vertex during each level-synchronous iteration of the search. Since in the worst case there can be $O(n)$ search iterations, their algorithm requires suboptimal $O(n^2+m)$ work complexity. This approach to solving graph traversal can be likened to using bubble sort to reorder data: easy to correctly

implement, but performs poorly.

Luo *et al.* presented an algorithm with asymptotically optimal $O(m+n)$ work complexity in 2010 [11]. In contrast to Harish and Narayanan, the implementation from Luo *et al.* keeps an explicit queue of vertices on the frontier of each search iteration. Since multiple threads need to write vertices to the end of the queue, atomic operations are used to prevent data races. The level-synchronous execution of the algorithm requires a global barrier between iterations. Since the CUDA programming model doesn't provide an explicit barrier between threads belonging to different Cooperative Thread Arrays (CTAs), the general method employed to ensure global synchronization is to simply launch multiple kernels. Having this type of synchronization at every level of the graph traversal becomes impractical because high-diameter graphs can require thousands of them, resulting in an expensive overhead. To alleviate this issue, Luo *et al.* use a hybrid approach based on the size of the frontier. If the frontier is small enough to be processed by one CTA, the global synchronization is superfluous and the CUDA `__syncthreads()` instrinsic can be used instead. For frontiers that are large enough to be processed by multiple CTAs but small enough to be processed by one CTA per Streaming Multiprocessor (SM) of the GPU (in this case, a GTX 280 with 30 SMs), an inter-block synchronization from the literature [13] can be used. Finally, for larger frontiers separate kernels are launched, forcing global synchronization. Overall, the implementation from Luo *et al.* shows multiple factors of speedup over that of Harish and Narayanan. The GPU algorithm from Luo *et al.* outperforms that of a dual-socket CPU on high-diameter, well-structured graphs representing road networks and grids; however, the CPU outperforms the GPU on scale-free graphs such as social networks [11].

Hong *et al.* present a BFS implementation that directly maps to the GPU architecture, showing up to 9x improvement over Harish and Narayanan for scale-free graphs [14]. Although their method still requires quadratic $O(n^2 + m)$ work complexity, their use of warp-centric programming addresses the common problem of workload imbalance among

threads. Workload imbalances for fine-grained BFS is most severe for *scale-free* graphs. A scale-free graph has a degree distribution that follows a power law, where a small number of vertices have a large number of outgoing edges and a large number of vertices have a small number of outgoing edges [15]. Since each thread is assigned to a different vertex in the current vertex frontier, the amount of work assigned to each thread depends on the outdegree of the vertex it is assigned. Hence, because scale-free graphs have a highly-varying distribution of outdegrees, scenarios arise where some threads inspect just one or two neighbors whereas others inspect up to every other vertex in the graph.

The above methods cannot fully maximize parallel performance because of either asymptotically inefficient algorithms, workload imbalances, or atomic operations that do not scale well. Merrill *et al.* present an parallelization that resolves these issues via the use of an efficient parallel prefix sum [10]. While atomic operations are useful on multi-core CPUs, they do not scale well to the thousands of threads that are launched by GPU kernels. Alternatively, parallel prefix sum allows threads to reorganize their sparse and imbalanced work assignments into dense and uniform tasks. Each thread will have a number of vertices that it must enqueue for the next level of the search. A prefix summation of these numbers will tell each thread where it should begin writing the vertices it will enqueue such that threads have their own reserved space for writing data without having to use atomic operations to safely "claim" a location in memory. Merrill *et al.* also provide techniques for removing duplicates from vertex frontiers [10]. This concept is especially important for GPU execution, since many threads can enqueue the same vertex before that vertex has been marked as visited. Each additional time that a vertex is enqueued presents a duplication of (redundant) work. This redundant work becomes especially troublesome for graphs representing grids, lattices, or meshes. These types of graphs, when searched in a level-synchronous breadth-first order, will have many incident edges from vertices in a frontier to the same vertex in the following frontier. To prevent this duplication of work, Merrill *et al.* maintain a cache

of recently inspected vertices in shared memory. Furthermore, they employ a hashing technique to reduce duplicates originating from the same warp. Overall, these techniques result in over 3.3 GTEPS (Billions of Traversed Edges per Second) on a diverse set of graphs. In contrast, prior single-node parallel CPU implementations could only achieve up to 1.3 GTEPS [10].

More recently, a hybrid approach to Breadth-First Search that is useful for low-diameter graphs [9] was presented. Although this implementation from Beamer *et al.* was designed for multi-core CPUs, it may be beneficial to GPU implementations of BFS as well. BFS is typically executed in a *top-down* fashion, meaning that vertices in the current vertex frontier search their adjacency lists for uninspected vertices. In contrast, Beamer *et al.* propose a *bottom-up* variation, in which vertices that have yet to be inspected search their parents to find if any of them are in the current frontier [9]. If they find any parent in the frontier the vertex can simply add itself to the next frontier. This alternative approach works best when a large portion of the vertices in the graph are in the current frontier, which will typically be the case for the middling iterations of a BFS on a scale-free or small world graph. Using an 8-core dual-socket Intel CPU, Beamer *et al.* present traversal rates that are over two times as fast as the single GPU implementation from Merrill *et al.* [9].

2.2 GPU Implementations of Graph Algorithms

Successful speedups of GPU graph traversal algorithms over that of parallel CPU algorithms have encouraged additional GPU implementations of graph algorithms. Unfortunately, GPU software design currently tends to be monolithic and inflexible in nature. Ideally, GPU algorithms should build on one another and be tunable for performance portability to different microarchitectures. Instead, developers tend build their algorithms without relying on previous work for assistance. Despite the monumental efforts required to construct correct and high performance graph algorithms on the GPU, speedups have been shown in the literature for algorithms such as strongly connected components, betweenness

centrality, single-source shortest paths, and others. The remainder of this Section reviews this literature in detail, pointing out current limitations in GPU graph analysis.

2.2.1 Strongly Connected Components

Barnat *et al.* provide an implementation for Strongly Connected Components in CUDA [16]. Strongly Connected Components (SCCs) are sets of maximal cycles. Determining whether or not a directed graph has a cycle at all is a subset of the SCC problem and is an important primitive for more complicated problems. For instance, the condensation of a directed graph is a mapping of that graph such that all of the vertices within each of its strongly connected components are collapsed into a single "super" vertex. An edge from a super vertex U to a super vertex V exists if and only if $\exists$ an edge (u, v) in the original graph such that $u \in U$ and $v \in V$. This coarsening of the graph can be used for applications such as graph partitioning on distributed systems or for the analysis of strongly connected networks that are otherwise too large to analyze in their nascent form [17].

Although Tarjan's algorithm is a well-known work-efficient sequential approach to finding the SCCs in a graph [18], a work-optimal $O(m + n)$ parallel approach has yet to be found [19]. Typical parallelizations of the SCC problem choose a pivot vertex at random and then find all vertices that are forward reachable and backward reachable from this pivot [20]. Vertices that belong to both the forward and backward reachable sets belong to an SCC that also contains the pivot. These vertices can then be removed from the graph, and the process can repeat. In fact, this process splits the graph into three independent sets from which pivots can be chosen: the set of vertices that were only forward reachable from the pivot, the set of vertices that were only backward reachable from the pivot, and the set of vertices that were neither forward nor backward reachable from the pivot [20]. Thus, in addition to the fine-grained parallelism used for performing the FW and BW searches, there is available coarse-grained parallelism for processing these independent partitions. Theoretically, each of these independent subpartitions can be further divided into even smaller

subgraphs; however, in pratice the number of concurrent subpartitions doesn't grow exponentially [19].

Another technique used to improve the performance of parallel SCC detection algorithms is *trimming*. If a vertex has either no incoming edges or no outgoing edges, then it belongs to its own SCC and can be removed, or "trimmed," from the graph [21]. Hence before choosing pivot vertices one can start from the perimeter of the graph and trim vertices to reduce the overall computational workload. Once a vertex is removed from the graph after its SCC is detected, edges that either reach or extend from that vertex can simply be ignored. Thus, trimming can be applied iteratively, as the trimming of one vertex could possibly allow for other vertices to be trimmed. When a vertex is trimmed, it no longer needs to be chosen as a pivot or needs to be found from another pivot's forward or backward traversals. Since the parallel inspection of vertex degrees is significantly faster than performing forward and backward traversals from these vertices, the use of trimming provides excellent performance [21]. The algorithm from Barnat *et al.* uses trimming although it inefficiently assigns threads to all unclassified vertices rather than the subset of these vertices that can potentially be trimmed. Furthermore the traversals implemented by Barnat *et al.* use inefficient $O(n^2 + m)$ graph traversals, which also hinders performance. Despite these weaknesses, the GPU implementation from Barnat *et al.* was able to outperform a sequential implementation of Tarjan's algorithm [16].

2.2.2 Betweenness Centrality

Betweenness Centrality (BC) was originally developed in the social sciences for classifying people who were central to social networks and could thus influence others by withholding information or altering it [22]. The metric attempts to distinguish the most influential vertices in a network by measuring the ratio of shortest paths passing through a particular vertex to the total number of shortest paths between all pairs of vertices. Intuitively, this ratio determines how well a vertex connects pairs of other vertices in the network. Formally,

the betweenness centrality score of a vertex v is defined as:

$$BC(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

where σ_{st} is the number of shortest paths between vertices s and t and $\sigma_{st}(v)$ is the number of those shortest paths that pass through v.

Having applications in community detection [23], power grid contingency analysis [24], and the study of the human brain [25], Betweenness Centrality has become a popular graph analytic in recent literature. Naïve implementations of Betweenness Centrality solve the all-pairs shortest-paths problem using the $O(n^3)$ Floyd-Warshall algorithm [26] and augment this result with path counting. Brandes improved upon this approach for sparse graphs with an algorithm that runs in $O(mn)$ time for unweighted graphs [27].

Due to the algorithm's costly $O(mn)$ time complexity, parallel implementations of betweenness centrality have been proposed on CPUs [28], GPUs [29, 30], heterogeneous architectures [31], and even special purpose supercomputers such as the Cray XMT [32].

Jia *et al.* discussed two fine-grained distributions of GPU threads to graph entities: *vertex-parallel* and *edge-parallel* [29]. The vertex-parallel approach assigns a thread to each vertex of the graph and that thread traverses all of the outgoing edges from that vertex. In contrast, the edge-parallel approach assigns a thread to each (directed) edge of the graph and that thread traverses that edge only.

The *GPU-FAN* package from Shi and Zhang was designed for the analysis of biological networks representing protein communications or genetic interactions [30]. Similar to the implementation from Jia *et al.*, GPU-FAN uses the edge-parallel method for load balancing across threads. The most significant difference between the two implementations is the distribution of CTAs to units of work.

The large amount of both coarse and fine-grained parallelism combined with the high memory bandwidth of the GPU allows the implementations from Jia *et al.* and Shi and Zhang to gain speedups of 7x-10x over existing sequential CPU methods.

2.2.3 Additional Algorithms

In addition to strongly connected components and betweenness centrality, a number of other graph algorithms have shown significant speedups over their CPU counterparts when parallelized on the GPU. Mendez-Lojo *et al.* show an implementation of Andersen's points-to analysis, a compiler analysis technique that determines what other variables a pointer variable may point to during the execution of a program [33]. This algorithm is especially difficult to parallelize since the structure of the graph representing these relationships between variables changes as the algorithm progresses. To overcome this challenge the authors developed a novel data structure based on wide sparse bit vectors. Their CUDA implementation achieved an average speedup of 7x over the state of the art CPU implementation and outperformed their own multi-CPU implementation on a system with 16 cores [33].

Davidson *et al.* provide work-efficient parallel algorithms to solve the Single-Source Shortest Paths (SSSP) problem [34]. Their analysis takes the tradeoff between algorithmic work and organizational overhead. Previously, a 40 core CPU system was able to achieve an edge traversal rate of 130 MTEPS while their GPU implementation achieved a maximum throughput of 350 MTEPS for an R-MAT graph of similar size [34]. The fact that this graph is an R-MAT graph is significant though, as these graphs exhibit large amounts of parallelism during graph traversals. Hence, the traversal rates for scale-free R-MAT graphs are a best-case scenario. Still, their implementation outperforms a sequential implementation of Dijkstra's algorithm for all classes of graphs other than road networks. Similar to graph traversal, the more available parallelism at each level synchronous step of the algorithm, the better the speedups the authors were able to obtain.

2.3 Limitations of Contemporary GPU Graph Analysis

Current GPU implementations of algorithms for network analysis are monolithic and inflexible. Domain experts in parallel programming for GPU architectures currently spend

months designing, testing, and tuning applications that can only be leveraged in a narrow scope [35]. These applications have limited reuse because even minor algorithmic changes can cause drastic and opaque differences in performance. Compared to CPU architectures, the hardware features of the GPU need to be explicitly handled by programmers. For example, shared memory is an explicit aspect of the underlying programming model. Supplying performance portability to multiple GPU microarchitectures can thus require extensive changes to existing algorithms such that programmers can capitialize on new architectural features. Social scientists and other users of graph analysis techniques should ideally be able to leverage the performance benefits offered by the GPU for their specific use cases without having to understand the complicated details of parallel programming models, the underlying architecture, or the performance-sensitive details of algorithm design. Such an environment would allow users to focus on their experiments and innovations in software. Furthermore, because GPU programming is still in a nascent state compared to that of conventional CPU software design, the available tools for debugging and profiling are limited, making the software development cycle very time consuming [35].

There are several libraries designed to increase programmer productivity and performance portability for parallel architectures, each of which has a slightly different center of attention. The most widely used of these libraries is *Thrust*, a parallel algorithms library that resembles the C++ Standard Template Library (STL) [7]. Thrust has multiple backends, such as CUDA, OpenMP, and Intel's Threading Building Blocks (TBB), and provides high-level abstractions to efficient implementations of parallel algorithms. Users can write simple, composable code in terms of these abstractions and obtain performance comparable to manually tuned code. From a GPU perspective, one limitation of Thrust is that its abstractions must be called from CPU threads on the host and cannot be called from GPU threads or CTAs on the device. Having some overlapping features with Thrust, *CUB* ("CUDA Unbound") is a library meant specifically for CUDA applications that provides

flexible, tunable implementations of common parallel primitives [8]. Since CUB is specifically targeted toward GPU backends, its algorithms can either be called from CPU threads on the host or from CTAs on the device.

Currently, libraries such as Thrust and CUB do not suffice for the composition of graph algorithms due to the limited set of primitives that they provide. In addition to common routines such as sorting, searching, and scanning, graph algorithms require traversal techniques and methods for performing operations on noncontiguous or data-dependent subsets of data. Furthermore, graph analysis is especially challenging because performance sensitivity comes from network structure in addition to algorithm design.

CHAPTER 3

SCALABLE AND HIGH PERFORMANCE GPU BETWEENNESS CENTRALITY

Network analysis is a fundamental tool for domains as diverse as compilers [33], social networks [36], and computational biology [25]. Real world applications of these analyses involve tremendously large networks that cannot be inspected manually. An example of a graph analytic that has found significant attention in recent literature is Betweenness Centrality (BC). Betweenness centrality has been used for finding the best location of stores within cities [37], studying the spread of AIDS in sexual networks [38], power grid contingency analysis [24], and community detection [39]. The variety of fields and applications in which this method of analysis has been employed shows that graph analytics require algorithmic techniques that make them performance portable to as many network structures as possible. Unfortunately, the fastest known algorithm for calculating BC scores has $O(mn)$ complexity for unweighted graphs with n vertices and m edges, making the analysis of large graphs challenging. Hence there is a need for robust, high performance graph analytics that can be applied to a variety of network structures and sizes.

GPUs provide high performance for regular, dense, and computationally demanding subroutines such as matrix multiplication. However, there has been recent success in accelerating irregular, memory-bound graph algorithms on GPUs as well [10, 33, 34]. Prior implementations of betweenness centrality on the GPU have outperformed their CPU counterparts, particularly on scale-free networks; however, they are limited in scalability to larger graph instances, use asymptotically inefficient algorithms that mitigate performance on high diameter graphs, and aren't general enough to be applied to the variety of domains that can leverage their results.

This chapter alleviates these problems by making the following contributions:

- We provide a work-efficient algorithm for betweenness centrality on the GPU that

works especially well for networks with a large diameter.

- For generality, we propose two algorithms that alternate between leveraging either the memory bandwidth of the GPU or the asymptotic efficiency of the work being done based on the structure of the graph being processed. The first of these approaches bases its decision on how significantly the size of the working set of vertices changes across iterations. The second is an on-line approach that uses a small amount of initial work from the algorithm to suggest which method of parallelism would be best for processing the remaining work.

- We implement our approach on a single GPU system, showing an average speedup of 2.71× across a variety of both real-world and synthetic graphs over the best previous GPU implementation. Additionally, our implementation attains near linear speedup on a cluster of 192 GPUs. Our single GPU approach achieves traversal rates up to 400 MTEPS (Millions of Traversed Edges per Second) while our multi-node approach achieves traversal rates exceeding 10 GTEPS (Billions of Traversed Edges per Second).

3.1 Background

3.1.1 Definitions

Let a graph $G = (V, E)$ consist of a set V of $n = |V|$ vertices and a set E of $m = |E|$ edges. A path from a vertex u to a vertex v is any sequence of edges originating from u and terminating at v. Such a path is a *shortest path* if its sequence contains a minimal number of edges. A Breadth-First Search (BFS) explores vertices of a graph by starting a "source" (or "root") vertex and exploring its neighbors. The neighbors of these vertices are then explored and this process repeats until there are no remaining vertices to be explored. Each set of inspected neighbors is referred to as a *vertex frontier* and the set of outgoing edges from a vertex frontier is referred to as an *edge-frontier*. The *diameter* of a graph is the length of the longest shortest path between any pair of vertices. A *scale-free* graph has a

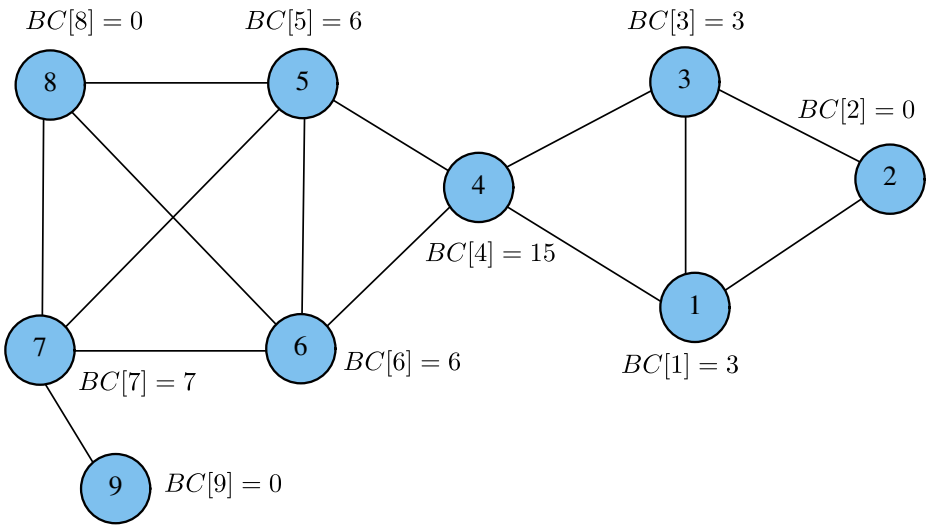


Figure 1: Example Betweenness Centrality scores for a small graph

degree distribution that follows a power law, where a small number of vertices have a large number of outgoing edges and a large number of vertices have a small number of outgoing edges [15]. Finally, a *small world* graph has a diameter that is proportional to the logarithm of the number of vertices in the graph [40]. In these networks every vertex can be reached from every other vertex by traversing a small number of edges.

3.1.2 Brandes's Algorithm

Betweenness centrality was originally developed in the social sciences for classifying people who were central to networks and could thus influence others by withholding information or altering it [22]. Formally, the Betweenness centrality of a vertex v is defined as:

$$BC(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2}$$

where σ_{st} is the number of shortest paths between vertices s and t and $\sigma_{st}(v)$ is the number of those shortest paths that pass through v.

Consider Figure 1. Vertex 4 is the only vertex that lies on paths from its left (vertices 5 through 9) to its right (vertices 1 through 3). Hence vertex 4 lies on all of the shortest paths between these pairs of vertices and has a high BC score. In contrast, vertex 9 does not

belong on a path between any pair of the remaining vertices in the graph and thus vertex 9 has a BC score of zero. Vertex 8 can be found on a path from vertex 5 to vertex 9; however, the shortest path from vertex 5 to vertex 9 instead goes through vertex 7. Since vertex 8 does not lie on any *shortest* paths between pairs of other vertices it also has a BC score of zero. Note that the scores reflected in Figure 1 treat a path from vertex u to vertex v as equivalent to a path from vertex v to vertex u since these paths are undirected. In other words, to avoid double counting the number of (undirected) shortest paths we divide the scores by two. One might also notice that the magnitude of BC values scales with the size of the network. For a fair comparison of BC values between vertices of two different graphs, a commonly used technique is to normalize the BC scores by their largest possible value [41]: $(n-1)(n-2)$. Such a comparison could be useful for comparing discrete slices of a network that changes over time [2].

Naïve implementations of Betweenness Centrality solve the all-pairs shortest-paths problem using the $O(n^3)$ Floyd-Warshall algorithm [26] and augment this result with path counting. Brandes improved upon this approach with an algorithm that runs in $O(mn)$ time for unweighted graphs [27]. The key concept of Brandes's approach is the *dependency* of a vertex v with respect to a given source vertex s:

$$\delta_s(v) = \sum_{w:v \in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)) \tag{3}$$

The recursive relationship between the dependency of a vertex and the dependency of its successors allows a more asymptotically efficient calculation of the centrality metric. Brandes's algorithm splits the betweenness centrality calculation into two major steps:

1. Find the number of shortest paths between each pair of vertices
2. Sum the dependencies for each vertex

We can redefine the calculation of BC scores in terms of dependencies as follows:

$$BC(v) = \sum_{s \neq v} \delta_s(v) \tag{4}$$

3.2 Prior GPU Implementations

Two well-known GPU implementations of Brandes's algorithm have been published within the last few years. Jia *et al.* [29] compare two types of fine-grained parallelism, showing that one is preferable over the other because it exhibits better memory bandwidth on the GPU. Shi and Zhang present *GPU-FAN* [30] and report a slight speedup over Jia *et al.* by avoiding data structure duplication and using a different distribution of threads to units of work. Both methods focus their optimizations on scale-free networks.

3.2.1 Vertex and Edge Parallelism

Jia *et al.* discussed two distributions of threads to graph entities: *vertex-parallel* and *edge-parallel* [29]. The vertex-parallel approach assigns a thread to each vertex of the graph and that thread traverses all of the outgoing edges from that vertex. In contrast, the edge-parallel approach assigns a thread to each edge of the graph and that thread traverses that edge only. In practice, the number of vertices and edges in a graph tend to be greater than the available number of threads so each thread sequentially processes multiple vertices or edges.

For both the shortest path calculation and the dependency accumulation stages the number of edges traversed per thread by the vertex-parallel approach depends on the out-degree of the vertex assigned to each thread. The difference in out-degrees between vertices causes a load imbalance between threads. For scale-free networks this load imbalance can be a tremendous issue, since the distribution of out-degrees follows a power law where a small number of vertices will have a substantial number of edges to traverse [15]. The edge-parallel approach solves this problem by assigning edges to threads directly.

Both the vertex-parallel and edge-parallel approaches from Jia *et al.* use an $O(n^2 + m)$ graph traversal that checks if each vertex being processed belongs to the current depth of the search rather than keeping an explicit queue of these vertices. For graphs with large diameters this method of graph traversal produces a large amount of unnecessary work in the form of branching overhead and accesses to global memory [10].

Furthermore, even for scale-free graphs the initial and final iterations of the traversal will have a comparably small vertex frontier and possibly a small number of edges to traverse for these iterations, depending on the connectivity of those vertices. In these cases the non-linear graph traversal can also be costly in terms of execution time.

Figure 2 illustrates this concept. Using the same graph shown in Figure 1, consider a Breadth-First Search starting at vertex 4. During the second iteration of the search, vertices 1, 3, 5, and 6 are in the vertex frontier, and hence their edges need to be inspected. The vertex-parallel method, shown in the top portion of Figure 2, distributes one thread to each vertex of the graph even though the edges connecting most of the vertices in the graph do not need to be traversed, resulting in wasted work. Also note that each thread is responsible for traversing a different number of edges (denoted by the small squares beneath each vertex), leading to workload imbalances. The edge-parallel method, shown in the middle portion of Figure 2, does not have the issue of load imbalance because each thread has one edge to traverse. However, this assignment of threads also results in wasted work because the edges that do not originate from vertices in the frontier do not need to be inspected in this particular iteration (but will be unnecessarily inspected during *every* iteration). Finally, the bottom portion of Figure 2 shows a work-efficient traversal iteration where each vertex in the frontier is assigned a thread. In this case only useful work is conducted although a load imbalance can exist among threads.

3.2.2 GPU-FAN

The *GPU-FAN* package from Shi and Zhang was designed for the analysis of biological networks representing protein communications or genetic interactions [30]. They report speedup ranging from 11% to 19% over the implementation from Jia *et al.* on a simulated scale-free network with the number of vertices varying from 10,000 to 50,000 and a varying preferential attachment of edges to vertices. Since these results are limited in scope, it is unclear as to which of these two implementations is preferable, especially for other types of networks such as small-world networks or high-diameter networks.

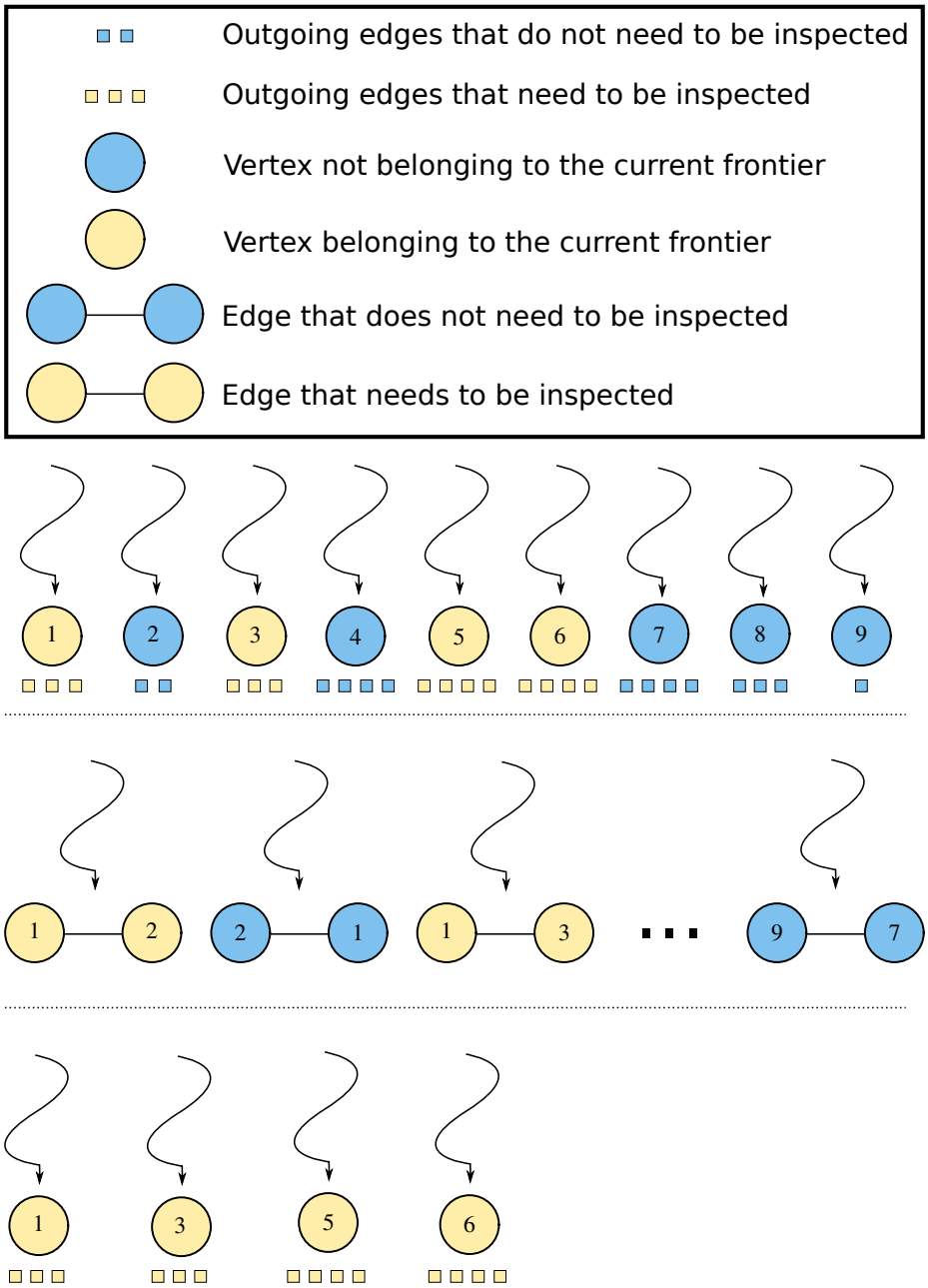


Figure 2: Illustration of the distribution of threads to units of work. Top: Vertex-parallel. Middle: Edge-parallel. Bottom: Work-efficient.

Like the previous implementation from Jia *et al.*, GPU-FAN uses the edge-parallel method for load balancing across threads. The most significant difference between the two implementations is the distribution of groups of threads (thread blocks using CUDA terminology) to units of work. The GPU-FAN package focuses only on fine-grained parallelism, using all threads from all thread blocks to traverse edges in parallel for one source vertex of the BC computation at a time. In contrast, the implementation from Jia *et al.* uses both coarse-grained and fine-grained parallelism. The threads within a block traverse edges in parallel while separate thread blocks each focus on the independent roots of the BC computation. This approach requires per-block data structures for the following variables:

- d, the BFS distance from the source vertex s to each vertex
- σ, the number of shortest paths from s to each vertex
- δ, the dependency accumulation of each vertex with respect to s
- P, the predecessor lists for each vertex with respect to s

The largest of these data structures is the predecessor list, which requires $O(m)$ space. Jia *et al.* showed that the best number of thread blocks to launch is equivalent to the number of Streaming Multiprocessors (SMs) on the GPU. Since the number of SMs that currently reside on GPU architectures is small, this additional storage requirement doesn't have a significant impact on scalability.

Another significant difference between these two implementations is that GPU-FAN uses $O(n^2)$ space for the predecessor list whereas Jia *et al.* use $O(m)$ space. Since each vertex besides the source vertex can have predecessors and since any of these vertices can have up to $O(n)$ predecessors based on the topology of the graph, using $O(n^2)$ space to store this information seems reasonable; however, an $O(m)$ array of boolean values can store this information more compactly. If edge number i represents an edge from vertex u to vertex v and u is the predecessor of v, then we can mark index i of an $O(m)$ predecessor

array as *true*. We will show that the choice of the $O(n^2)$ data structure for the predecessor array severely limits the scalability of this algorithm in Section 3.4. It should also be noted that GPU-FAN's approach also has the $O(n^2 + m)$ graph traversal issues mentioned in the analysis of the algorithm from Jia *et al.*

3.3 Methodology

3.3.1 Work-efficient Approach

Algorithm 1: Work-efficient Betweenness Centrality Local Variable Initialization

```
1  for v ∈ V do in parallel
2      if v = s then
3          d[v] ← 0
4          σ[v] ← 1
5      else
6          d[v] ← ∞
7          σ[v] ← 0
8      δ[v] ← 0
9  Q_curr[0] ← s ; Q_curr_len ← 1
10 Q_next_len ← 0
11 S[0] ← s ; S_len ← 1
12 ends[0] ← 0 ; ends[1] ← 1 ; ends_len ← 2
13 shared depth ← 0
```

Taking note of the issues mentioned in the previous section, we now present the basis for our work-efficient implementation of betweenness centrality on the GPU. Our approach leverages optimizations from the literature in addition to our own novel techniques. Algorithm 1 shows how we initialize local variables before each of the n shortest path calculations and dependency accumulations. The first way in which our implementation differs from prior GPU implementations is that we discard the predecessor array. Since all of the other local data structures require $O(n)$ memory, we reduce the space complexity of our local data structures from $O(m)$ to $O(n)$. This removal of space comes at the cost of additional computation, but does not change the overall computational complexity of the algorithm. In the dependency accumulation stage, rather than traversing the predecessors directly, all

of the neighbors of a vertex are instead traversed. This technique, known as the *neighbor traversal* approach from Green and Bader [42], not only reduces storage requirements to enhance the scalability of the algorithm, but also has been shown to generate speedups on multi-core systems.

Algorithm 1 shows a second major difference between the work-efficient approach and prior GPU implementations: the use of explicit queues for graph traversal. We initialize Q_{curr}, Q_{next}, and their respective lengths for the shortest path calculation stage. Since levels of the graph are processed in parallel we use two queues to distinguish vertices that are in the current level of the search (Q_{curr}) from vertices that are to be processed during the next level of the search (Q_{next}). For the dependency accumulation stage we initialize S and its length. In this case, we need to keep track of vertices at all levels of the search and hence we only use one data structure to store these vertices. To distinguish the sections of S that correspond to each level of the search we use the $ends$ array, where $ends_{len} = max_{v \in V, d[v] \neq \infty}\{d[v]\} + 1$ at the end of the traversal. Vertices corresponding to depth i of the traversal are located from index $ends[i]$ to index $ends[i+1] - 1$ of S. This usage of the $ends$ and S arrays is comparable to the arrays used to store the graph in CSR format.

A work-efficient shortest path calculation stage is shown in Algorithm 2. Iterations of the while loop correspond to the traversal of depths of the graph. The parallel for loop in Line 3 assigns one thread to each element in the queue such that edges from other portions of the graph aren't unnecessarily traversed. The atomic Compare and Swap (CAS) operation on Line 5 is used to prevent multiple insertions of the same vertex into Q_{next}. This restriction allows us to safely allocate $O(n)$ memory for Q_{next} instead of $O(m)$ in the case that duplicate queue entries are allowed. Since we only require one thread for each element in Q_{curr} rather than one thread for every vertex or edge in the graph, this atomic operation experiences limited contention and thus doesn't significantly reduce performance. Merrill *et al.* show that a prefix sum can be used to have threads cooperatively add elements to the queue [10], reducing the contention for the atomic instruction that we use on Line 6.

Algorithm 2: Work-efficient Betweenness Centrality Shortest Path Calculation

```
1  Stage 1: Shortest Path Calculation
2  while true do
3      for v ∈ Q_curr do in parallel
4          for w ∈ neighbors(v) do
5              if atomicCAS(d[w], ∞, d[v] + 1) = ∞ then
6                  t ← atomicAdd(Q_next_len, 1)
7                  Q_next[t] ← w
8              if d[w] = d[v] + 1 then
9                  atomicAdd(σ[w], σ[v])
10     barrier()
11     if Q_next_len = 0 then
12         depth ← d[S[S_len − 1]] - 1
13         break
14     else
15         for tid ← 0 ... Q_next_len − 1 do in parallel
16             Q_curr[tid] ← Q_next[tid]
17             S[tid + S_len] ← Q_next[tid]
18         barrier()
19         ends[ends_len] ← ends[ends_len − 1] + Q_next_len
20         ends_len ← ends_len + 1
21         Q_curr_len ← Q_next_len
22         S_len ← S_len + Q_next_len
23         Q_next_len ← 0
24         barrier()
```

However, their work focuses explicitly on BFS and differs from ours in that they are using all SMs of the GPU to perform one high-performance graph traversal whereas we are performing many graph traversals on each SM independently. In our tests we found that the overhead of the prefix sum was too large because the number of elements to sum on each SM is Q_{curr_len}, which is $O(n)$ in the worst case. When all SMs can contribute to these sums, as is the case in Merrill's work, this overhead is significantly reduced because each SM can independently do its portion of the sum in parallel and then the local sums can be reduced into a global sum. For our work on betweenness centrality all SMs have to perform their own sums (of the same number of elements) independently.

The conditional on Line 11 checks to see if the queue containing vertices for the next depth of the search is empty; if so, the search is complete, so we break from the outermost while loop. Otherwise, we transfer vertices from Q_{next} to Q_{curr}, add these vertices to the end of S for the dependency accumulation, and do the appropriate bookkeeping to set the lengths of these arrays.

Algorithm 3: Work-efficient Betweenness Centrality Dependency Accumulation

```
1  Stage 2: Dependency Accumulation
2  while depth > 0 do
3      for tid ← ends[depth] ... ends[depth + 1] − 1 do in parallel
4          w ← S[tid]
5          dsw ← 0
6          sw ← σ[w]
7          for v ∈ neighbors(w) do
8              if d[v] = d[w] + 1 then
9                  dsw ← dsw + (sw / σ[v])(1 + δ[v])
10         δ[w] ← dsw
11     barrier()
12     depth ← depth − 1
```

Algorithm 3 shows a work-efficient dependency accumulation. In addition to using the neighbor traversal approach, we are also able to eliminate the use of atomics by checking *successors* rather than the predecessors of each vertex. Since vertices at the end of the

BFS tree by definition have no successors, we start the dependency accumulation one level closer to the root of the tree (see Line 12 of Algorithm 2). Furthermore, since the source vertex of the BFS tree does not contribute to its own BC score, there is no need to perform any work when $depth = 0$. The technique of checking successors was developed by Madduri *et al.* for their implementation of betweenness centrality on the Cray XMT supercomputer [32]. Rather than having multiple vertices that are currently being processed in parallel update the dependency of their common ancestor atomically, the ancestor can update itself based on its successors without the need for atomic operations. Interestingly, an edge-parallel implementation the successor approach would still require atomic operations because multiple threads could be assigned to the same ancestor.

Note that the parallel for loop in Line 3 of Algorithm 3 assigns threads only to vertices that need to accumulate their dependency values; this is where the bookkeeping done to keep track of separate levels of the graph traversal in the *ends* array comes to fruition. Rather than naïvely assigning a thread to each vertex or edge and checking to see if that vertex or edge belongs to the current depth we instead can instantly extract vertices of that depth since they are a consecutive block of entries within S. This strategy again prevents unnecessary branch overhead and accesses to global memory that are made by previous implementations.

3.3.2 Hybrid Approach

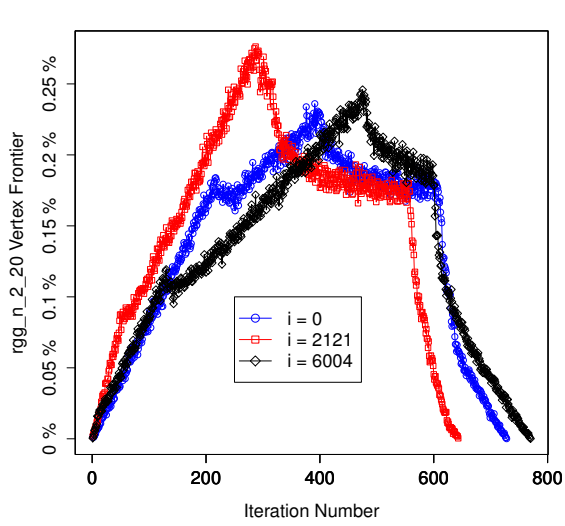
The major drawback of the approach outlined in the previous section is the potential for significant load imbalance between threads. Although our approach efficiently assigns threads to units of useful work, the distribution of edges to threads is entirely dependent on the structure of the graph. Our approach is significantly faster than other methods on graphs with a large diameter because such graphs tend to have a more uniform distribution of outdegree. On scale-free or small world graphs, however, the algorithm outlined in the previous section does not improve performance. For these graphs there are iterations of the graph traversal that require the inspection of a high percentage of edges in the graph. For

Table 1: Correlation of vertex and edge frontier sizes with execution time for three randomly selected roots of different types of graphs. The size of the vertex frontier correlates positively with execution time regardless of the root or structure of the graph.

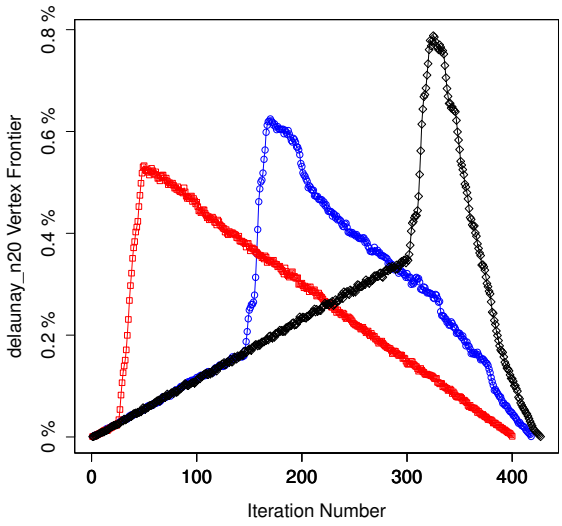
Graph	Root	$\rho_{v,t}$	$\rho_{e,t}$
rgg_n_2_20	0	**0.950**	0.950
	2121	**0.978**	0.976
	6004	**0.981**	0.980
delaunay_n20	0	**0.990**	0.990
	2121	**0.995**	0.995
	6004	**0.995**	0.995
kron_g500-logn20	0	**0.798**	0.093
	2121	**0.704**	0.195
	6004	**0.936**	-0.096
luxembourg.osm	0	**0.885**	0.883
	2121	**0.898**	0.892
	6004	**0.910**	0.907
smallworld	0	**0.967**	0.970
	2121	**0.989**	0.998
	6004	**0.995**	0.996

these iterations the load balance and high memory-throughput of the edge-parallel method combined with the fact that most of the edges inspected in fact need to be inspected means that the edge-parallel method is preferable to the work-efficient method. Based on this result we propose a hybrid approach that chooses between the edge-parallel and work-efficient methods based on the structure of the graph. Rather than preprocessing the graph to attempt to determine if it can be classified as a scale-free or small world graph, we implement our hybridization at a finer granularity: each iteration of the search.

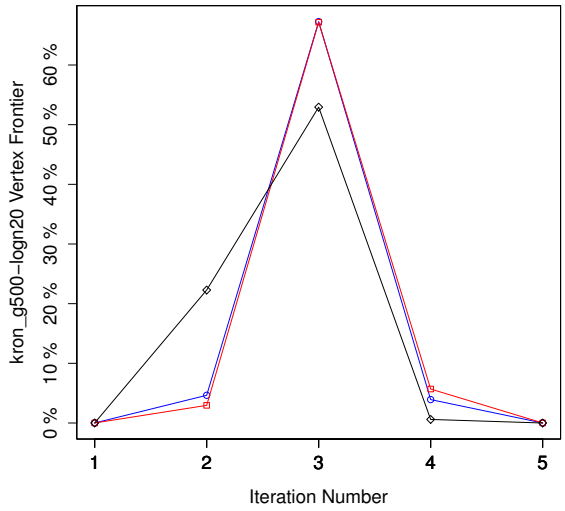
Figure 3 illustrates our rationale behind this decision. Each sub-figure shows how the vertex frontier evolves for three randomly chosen source vertices within a graph. Note that the axes of the sub-figures are on different scales to appropriately show trends in the frontiers. The sub-figures represent different classifications of graphs: meshes, road networks, scale-free, and small-world graphs. More information on these graphs can be found in Table 2. Although the position of the source vertex plays an important role in precisely how the vertex frontier changes with search iteration, we can see that the general sizes and changes in size of the vertex frontier across iterations of the search are more dependent



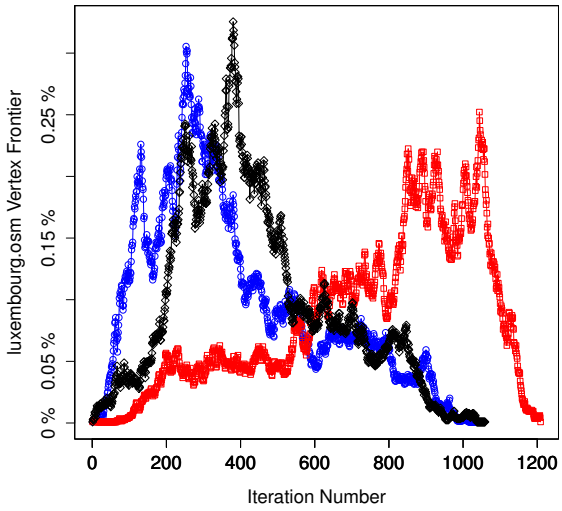
(a) rgg_n_2_20



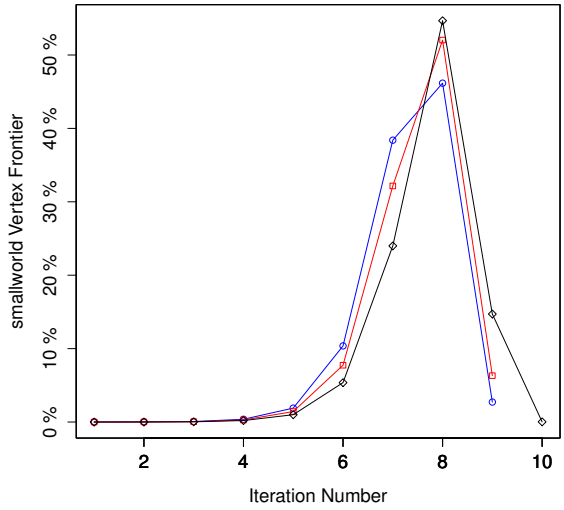
(b) delaunay_n20



(c) kron_g500-logn20



(d) luxembourg.osm



(e) smallworld

Figure 3: Evolution of vertex frontiers (as a percentage of total vertices) for different classifications of graphs

on the overall structure of the graph. For high-diameter graphs such as *rgg_n_2_20*, *delaunay_n20*, and *luxembourg.osm* (Figures 3a, 3b, and 3d respectively), the vertex frontier grows gradually and is always a small portion of the total number of vertices in the graph. For graphs with a smaller diameter such as *kron_g500-logn20* and *smallworld* (in Figures 3c and 3e), the vertex frontier grows large after just a few iterations and contains over half of the total number of vertices in the graph at its peak. Both small-world and scale-free graphs tend to exhibit this behavior. Although the edge-frontier accurately estimates the amount of necessary work for a given iteration, we found that the size of the vertex frontier correlates quite well with the execution time of an iteration. For instance, see Table 1. For the same source vertices whose vertex frontiers were plotted in Figure 3 we record the elapsed time of each iteration (using the work-efficient method) and calculate two correlation coefficients: $\rho_{v,t}$, which is the correlation between the size of the vertex frontier and elapsed time, and $\rho_{e,t}$, which is the correlation between the size of the edge frontier and elapsed time. The correlation of the vertex frontier with execution time is more robust to changes in source vertex or graph structure. This is an important result because we have the size of the vertex frontier at every iteration (since we keep an explicit queue). Obtaining the size of the edge frontier, in contrast, would require additional computation and fetches to memory.

Intuitively, for large vertex frontiers, the edge-parallel approach is favorable because of its memory throughput whereas for small vertex frontiers the work-efficient approach is favorable because the number of edges that will be traversed is significantly smaller than the total number of edges in the graph. While it is clear that the work-efficient approach is the best choice for all search iterations for graphs with a high diameter, the hybrid approach is especially useful for scale-free and small world graphs. These graphs contain search iterations that can have either a small or large vertex frontier (compared to the total number of vertices), which means that the work-efficient approach could be best for some iterations while the edge-parallel approach could be best for others. Graph structures that prefer

the work-efficient method tend to have consistent vertex frontiers, each of which contain a small percentage of vertices in the graph. Using the edge-parallel approach for any iteration would be wasteful for these classes of input. In contast, when the edge-parallel approach is favored, there will be some search iterations that have a comparably small amount of required work. For example, the initial iteration of the search simply expands the root vertex itself. If the root vertex is not a high degree vertex the work required (and available parallelism) for this iteration will be limited. Ideally, we would use the work-efficient method to process this iteration.

Initially, we measured the size of the vertex frontier at a given iteration and compared it to the total number of vertices in the graph. If this ratio exceeded a threshold, then the edge-parallel approach would be used. Otherwise, the work-efficient approach was used. The problem with this approach is that the percentage of vertices found in the queue can have tremendously different implications for different scales of graphs, even if those graphs have a similar classification or structure. For instance, a road network consisting of 1,000,000 vertices would be much more likely to use the edge-parallel approach than a road network consisting of 1,000 vertices, even though neither network would benefit from this approach.

Similarly, if we were to instead use the absolute size of the frontier as a threshold we would also observe undesired outcomes. The fact that the size of the vertex frontier has crossed a certain threshold gives no information about *how* it crossed that threshold. If the threshold is close to the size of the graph, then the edge-parallel approach would be performing mostly useful edge traversals and is likely to be the better choice; however in the case that the threshold is much smaller than the size of the graph the edge-parallel approach would perform many unnecessary edge traversals and the work-efficient approach would instead be the better choice.

We instead would like to detect when the frontier is significantly changing from one iteration to another as this information tells us when our strategy should change. If the

frontier is large and has not significantly changed (i.e. is still large), then we should continue to use the edge-parallel method to leverage its memory throughput. Conversely, if the frontier is small and has not significantly changed then we should continue to use the work-efficient method. Hence, we only need to reconsider our parallelization strategy when the size of the frontier changes from small to large or vice versa. Once we have decided that we the frontier has changed in this way, we can select the proper strategy based on the size of the frontier that is to be processed during the next iteration of the traversal.

Algorithm 4: Hybrid method for selecting parallelization strategy

```
1 Q_change = abs(Q_next_len − Q_curr_len)
2 if Q_change > α then
3     if Q_next_len > β then
4         //Choose edge-parallel method
5     else
6         //Choose work-efficient method
```

Algorithm 4 describes this process. The variable Q_{change} represents the change in size of the vertex frontier. If this value is less than or equal to the parameter α, then we continue to use the same strategy. Otherwise, we noticed that the size of the vertex frontier has substantially changed and that we should reconsider our strategy. If the number of vertices to be processed during the next iteration (Q_{next_len}) is larger than the parameter β, then we choose the edge-parallel method. Otherwise, we opt for the work-efficient method. In our experiments we found the values of 768 and 512 were the best choices for α and β, respectively. Although we were able to obtain favorable results in comparison to prior methods using these choices of α and β for all of the graphs tested, poor selection of these parameters can, in general, have a significant impact on performance. We initially start our calculations with the work-efficient method for two reasons. Firstly, the initial vertex frontier is simply the root itself, and for sparse graphs this vertex is unlikely to be heavily connected to the rest of the graph. Secondly, in the case that the edge-parallel method is preferred, using the work-efficient method shows a 2.2x slowdown in worst case for the

input sets that we tested. However, in the case that the work-efficient method is preferred, using the edge-parallel method can show greater than a 10x reduction in speed. Thus, incorrectly choosing the edge-parallel method is more costly than incorrectly choosing the work-efficient method.

3.3.3 Sampling

The exact computation of betweenness centrality computes a BFS for each vertex in the graph. Since all of these searches are independent, they can be executed in parallel. For large graphs of interest, there often are fewer available parallel resources than vertices in the graph. For example, the Titan supercomputer at Oak Ridge National Laboratory has 18,688 GPUs and it is ranked second on the June 2014 TOP500 list [43, 44]. For graphs whose vertices mostly belong to one large connected component, the amount of time to process each root is roughly equivalent, as the same number of edges need to be traversed for each root. Therefore the amount of time required to process k vertices is roughly k times the time required to process one vertex [45].

Algorithm 5: Sampling method for selecting parallelization strategy

```
   Input: Set of n_samps connected component sizes (keys)
1  sort(keys)
2  barrier()
3  if keys[n_samps/2] < γ * log2(n) then
4      //Choose edge-parallel method
```

Using the above analysis, an estimate of the average size of the connected components within the graph (and thus the preferred method of parallelism) is obtained by processing a small subset of the vertices and storing the maximum distance of the BFS from these vertices. Essentially this method willingly computes a small number of source vertices using a potentially unsatisfactory method of parallelism and uses this result to ensure that the desired method of parallelism is used to compute the remaining source vertices. Algorithm 5 shows how this method is implemented. For our implementation we set n_{samps} to 512 and process these vertices using the work-efficient method. For these vertices, we

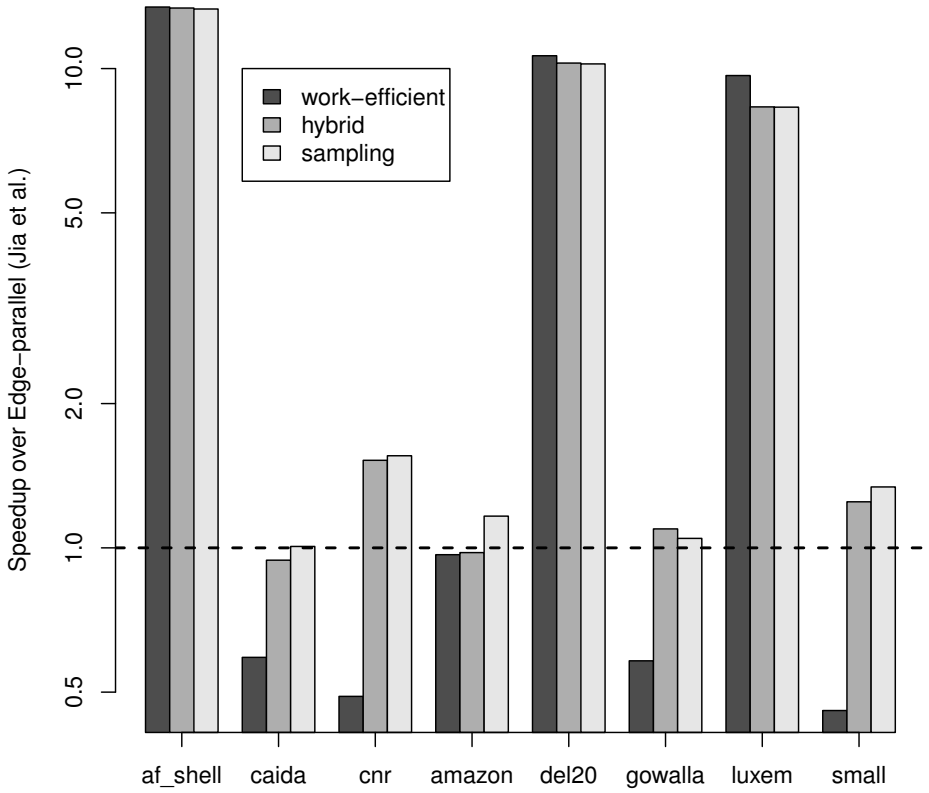


Figure 4: Comparison of Work-Efficient, Hybrid, and Sampling methods

record the maximum depth of each of their BFS traversals and take the median of this set as our estimate. We prefer the median because it is an unbiased estimator and less sensitive to outliers. If this median is sufficiently small then it is likely that our graph is a small-world or scale-free graph and thus we should switch to using the edge-parallel approach. Again, the work-efficient method is chosen by default; Algorithm 5 helps us determine whether or not to deviate from that initial strategy. In our experiments we found the value of $\gamma = 4$ to be best.

The advantage to the sampling approach is that it leverages completed work to come to its decision rather than preprocessing the graph or doing calculations that do not directly advance the progress of the program. The drawback of the sampling approach is that it does not switch its strategy at the granularity of a search iteration like the hybrid method does. Given our analysis of Figure 3, we recall that when the work-efficient method is preferred, it is preferred for all iterationsof the traversal. Hence, we only need the search iteration

Table 2: Graph datasets used for this study

Graph	Vertices	Edges	Max degree	Diameter	Description
af_shell9 [46]	504,855	8,542,010	39	497	Sheet metal forming
caidaRouterLevel [47]	192,244	609,066	1,071	25	Router-level topology
cnr-2000 [47]	325,527	2,738,969	18,236	33	Web crawl
com-amazon [48]	334,863	925,872	549	46	Amazon co-purchasing
delaunay_n20 [47]	1,048,576	3,145,686	23	444	Random triangulation
kron_g500-logn20 [49]	1,048,576	44,619,402	131,503	6	Kronecker
loc-gowalla [50]	196,591	1,900,654	29,460	15	Geosocial
luxembourg.osm [47]	114,599	119,666	6	1,336	Road map
rgg_n_2_20 [51]	1,048,576	6,891,620	36	864	Random geometric
smallworld [40]	100,000	499,998	17	9	Small world phenomenon

level of granularity for choosing our method of parallelism when the edge-parallel method is best for a given network. To avoid using the edge-parallel method for iterations of the search that have trivial amounts of work we simply check the size of the queue to make sure it is sufficiently large. If it is, we proceed with the edge-parallel method; otherwise, we revert to the work-efficient method. We perform this check at every search iteration when the sampling method chooses the edge-parallel approach. Similar to the use of β in the hybrid approach, the sampling approach requires the vertex frontier to contain at least 512 elements to use the edge-parallel method. This parameter is designed to scale with the architecture rather than the size or structure of the graph.

Figure 4 provides a comparison of the various parallelization methods discussed in this chapter to the edge-parallel method from Jia et. al [29]. For road networks and meshes (*af_shell, del20, luxem*) all of the methods outperform the edge-parallel method by about 10×. The amount of unnecessary work performed by the edge-parallel method for these graphs is severe. Note that the work-efficient method outperforms both the hybrid and sampling methods for these graphs. The latter methods require either additional computation or overhead for deciding which method of parallelism to use; this difference in performance is essentially the cost of generality. For the remaining graphs (scale-free and small-world graphs) using the work-efficient method alone performs slower than the edge-parallel method whereas the hybrid and sampling methods are either the same or slightly

better. In these cases we see the advantage of choosing our method of parallelization at the granularity of a search iteration. If information about the structure of the graph is known a priori, then the value of α can be adjusted accordingly; however, given no information at all, the sampling approach slightly outperforms the hybrid approach overall. The sampling approach deduces the structure of the graph based on information extracted from completed a small portion of useful work whereas the hybrid method estimates the structure by using the parameters α and β in addition to vertex frontier sizes.

3.4 Results

3.4.1 Experimental Setup

Single-node GPU experiments were implemented using the CUDA 6.0 Toolkit. The CPU is an Intel Core i7-2600K processor running at 3.4 GHz with an 8 MB cache and 16 GB of DRAM. The GPU is a GeForce GTX Titan that has 14 Streaming Multiprocessors (SMs) and a base clock of 837 MHz. The Titan has 6 GB of GDDR5 memory and is a CUDA compute capability 3.5 ("Kepler") GPU.

Multi-node experiments were run on the Keeneland Initial Delivery System (KIDS) [52]. KIDS has two Intel Xeon X5660 CPUs running at 2.8 GHz and three Tesla M2090 GPUs per node. Nodes are connected by an Infiniband QDR network. The Tesla M2090 has 16 SMs, a clock frequency of 1.3 GHz, 6 GB of GDDR5 memory, and is a CUDA compute capability 2.0 ("Fermi") GPU.

We compare our techniques to both GPU-FAN [30] and Jia *et al.* [29] when possible, using their implementations that have been provided online[1]. The graphs used for these comparisons are shown in Table 2. These graphs were taken from the 10th DIMACS Challenge [47], the University of Florida Sparse Matrix Collection [46], and the Stanford Network Analysis Platform (SNAP) [48, 50]. These benchmarks contain both real-world and randomly generated instances of graphs that correspond to a wide variety of practical applications and network structures. Although numerous approaches for approximating

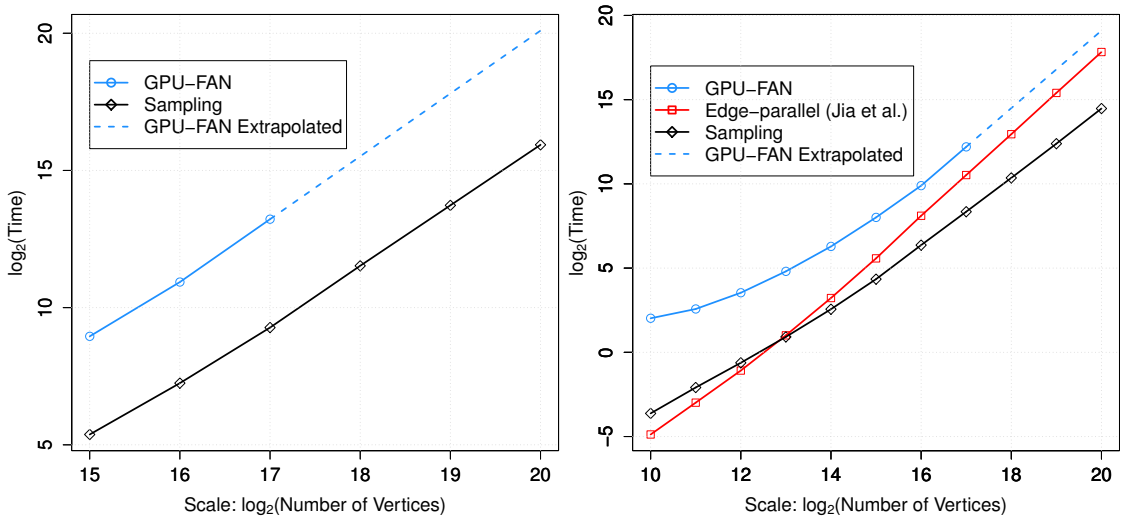
[1] Our implementation is available at https://github.com/Adam27X/hybrid_BC

Betweenness Centrality have been proposed [53, 54], we focus our attention on its exact computation, noting that our techniques can be trivially adjusted for approximation.

3.4.2 Scaling

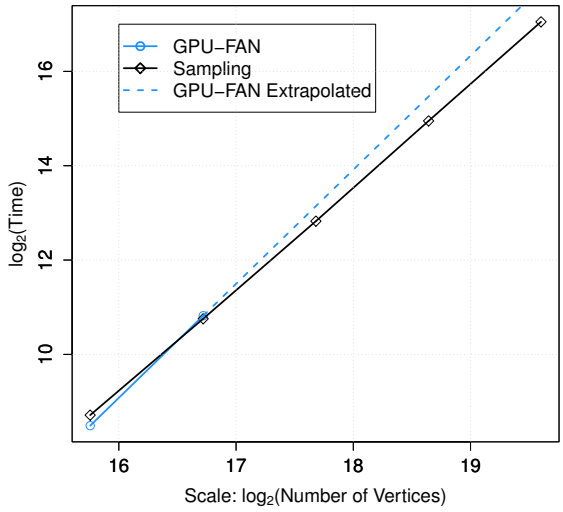
First we compare how well our algorithm scales with graph size for three different types of graphs. Since the implementation of Jia *et al.* cannot read graphs that contain isolated vertices, we were unable to obtain results using this reference implementation for the random geometric (*rgg*) and simple Kronecker (*kron*) graphs. Additionally, since the higher scales caused GPU-FAN to run out of memory, we simply extrapolated what we would expect these results to look like from the results at lower scales (denoted by dotted lines). Note that from one scale to the next the number of vertices and number of edges both double.

Noting the log-log scale on the axes, we can see from Figure 5a that the sampling approach outperforms the algorithm from GPU-FAN by over 12x for all scales of *rgg*. It is interesting to note that the sampling approach only takes slightly more time than GPU-FAN when the sampling approach processes a graph four times as large. For the *delaunay* mesh graphs in Figure 5b we can see that the edge-parallel method and the sampling approach both outperform GPU-FAN for all scales. The edge-parallel approach even outperforms the sampling approach for graphs containing less than 10,000 vertices; however, it should be noted that these differences in timings are trivial as they are on the order of milliseconds. As the graph size increases the sampling method clearly becomes dominant and the speedup it achieves grows with the scale of the graph. Again, the sampling approach can handle a graph with a million vertices faster than the previous approaches can handle a graph that is only half as large. Finally, we compare the sampling approach to GPU-FAN for *kron* in Figure 5c. Although GPU-FAN is marginally faster than the sampling approach for the smallest scale graph we can see that the sampling approach is best at the next scale and the trend shows the amount by which the sampling approach is best grows with scale. Furthermore, neither of the previous implementations could support this type of graph at larger scales whereas the sampling method can support even larger scales.



(a) rgg

(b) delaunay



(c) kron

Figure 5: Scaling by problem size for three different types of graphs

Table 3: Performance of edge-parallel and sampling methods for various graphs. Results are in MTEPS (Millions of Traversed Edges per Second).

Graph	Edge-parallel	Sampling	Speedup
af_shell9	18.00	239.66	13.31x
caidaRouterLevel	180.98	182.21	1.01x
cnr-2000	141.75	220.64	1.56x
com-amazon	109.72	127.79	1.16x
delaunay_n20	14.19	145.09	10.23x
loc-gowalla	209.56	219.31	1.05x
luxembourg.osm	4.74	39.42	8.31x
smallworld	297.48	398.63	1.34x
Average	**2.71x Geometric Mean Speedup**		

3.4.3 Benchmarks

For graph algorithms, standard metrics such as FLoating-point Operations Per Second (FLOPs) are not accurate indicators of performance because most of their processing time is spent accessing memory [55]. One alternative metric to FLOPs used to measure the performance of data-intensive algorithms is the number of Traversed Edges per Second (TEPS). For the exact computation of betweenness centrality that is considered in this chapter, the number of TEPS has been defined as [45]:

$$TEPS_{BC} = \frac{mn}{t} \tag{5}$$

In this equation, n is the number of vertices, m is the number of (undirected) edges, and t is the execution time of the BC computation.

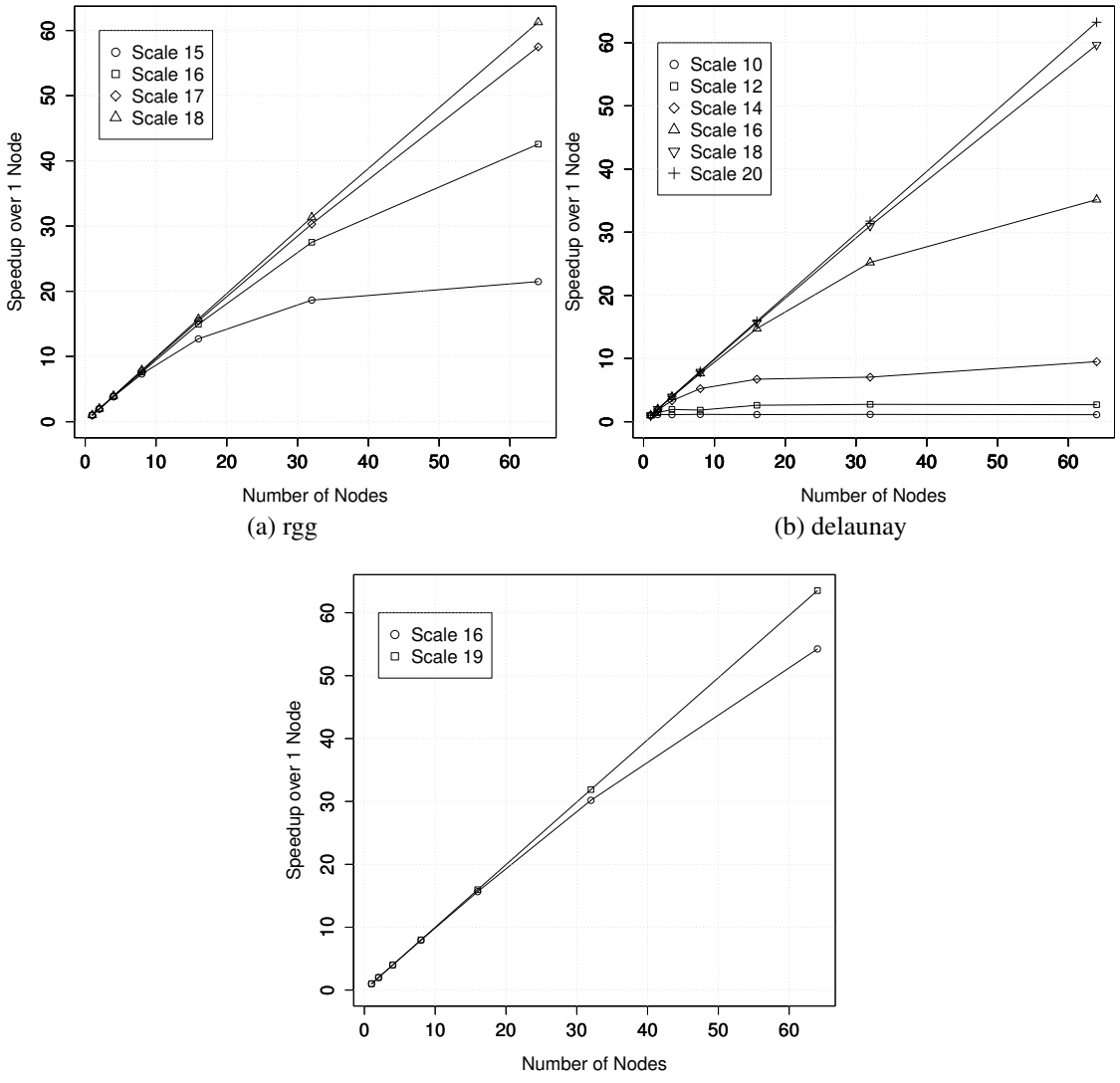
Table 3 compares sampling results to Jia *et al.* only because the graphs tested are too large to be analyzed by GPU-FAN. Results are reported in Millions of Traversed Edges Per Seconds (MTEPS). In the most extreme case, the edge-parallel approach requires more than two and half days to process the *af_shell9* graph while the sampling approach cuts this time down to under five hours. Similarly, the edge-parallel approach takes over 48 minutes to process the *luxembourg.osm* road network whereas the sampling approach requires just 6 minutes. We can see that the sampling approach achieves approximately 40 MTEPS for all of these graphs whereas the edge-parallel method has particularly low MTEPS for

high-diameter graphs. The TEPS metric described above only accounts for edges that need to be traversed by the algorithm (i.e. useful work). Since the edge-parallel approach naïvely traverses every edge for every BFS iteration of every root, the number of useful edge traversals per unit time is overcome by futile edge inspections. Overall, sampling performs 2.71x faster on average than the edge-parallel approach.

3.4.4 Multi-GPU Experiments

Although our approaches leverage both coarse and fine-grained parallelism there is still more available parallelism than can be handled by a single GPU. Our methods easily extend to multiple GPUs as well as multiple nodes. We extend the algorithm by distributing a subset of roots to each GPU. Since each root can be processed independently in parallel, we should expect close to perfect scaling if each GPU has a sufficient (and an evenly distributed) amount of work. For graphs that have one very large connected component the amount of work to perform (in terms of the number of edges to traverse) will be equivalent for each root and thus, each GPU if the number of GPUs divides evenly into the number of source vertices. For graphs that have a larger number of connected components an imbalance between GPUs is of course more probable; however, since each GPU processes hundreds of source vertices (or more) it is highly likely for each GPU to process source vertices from each connected component.

Although the local data structures for each root are independent (and thus only need to reside on one GPU), we replicate the data representing the graph itself across all GPUs to eliminate communication bottlenecks. Once each GPU has its local copy of the BC scores these local copies are accumulated for all of the GPUs on each node. Finally, the node-level scores are reduced into the global BC scores by a simple call to $MPI\_Reduce()$. Figure 6 shows how well our algorithm scales out to multiple GPUs for *delaunay*, *rgg*, and *kron* graphs. It shows that linear speedup is easily achievable if the problem size is sufficiently large (i.e. if there is sufficient work for each GPU). Looking at the *delaunay* 64 node case specifically, it appears that the graph needs at least 2^{18} vertices to achieve



(a) rgg (b) delaunay

(c) kron

Figure 6: Multi-GPU scaling by number of nodes for various graph structures. Each node contains three GPUs.

Table 4: Multi-node performance for various graphs. Results are in GTEPS (Billions of Traversed Edges per Second).

Graph	64 Nodes (GTEPS)	Speedup over 1 Node
rgg_n_2_20	8.25	63.34x
delaunay_n20	9.37	63.24x
kron_g500-logn20	24.13	63.75x

near linear speedups. Since each node contains 3 GPUs, this allocates at least 1350 root vertices to each GPU. Furthermore, since the *delaunay* graphs have a particularly small ratio of edges to vertices, these particular graphs need *more* work per GPU than denser graphs that typically occur in real-world problems. Hence linear speedups are achievable at even smaller scales of graphs for denser network structures. For instance, using 64 nodes provides about a 35× speedup over a single node for scale 16 *delaunay* graph whereas using the same number of nodes at the same scale for *rgg* and *kron* graphs provides over 40× and 50× speedups respectively. The scaling behavior seen in Figure 6 is not unique to these graphs because of the vast amount of coarse-grained parallelism offered by the algorithm. For graphs of large enough size this scalability can be obtained independently of network structure.

Table 4 shows TEPS rates for our 64 node (192 GPU) implementation. Results are reported in Billions of Traversed Edges per Second (GTEPS). For each graph classification we see almost perfect linear speedup over 1 node (3 GPUs). The notably better TEPS rate for the Kronecker graph occurs because this graph tends to have more isolated vertices than real world graphs (or even other synthetic graphs) due to how it is randomly generated. Since the calculation for TEPS implicitly assumes that all vertices belong to one connected component, the reported TEPS value for *kron_g500-logn20* is inflated. Nevertheless, over 75% of the vertices for this graph are not isolated, and adjusting for this factor still results in approximately 18 GTEPS for this graph. The reason that this adjusted value is still greater than the TEPS values for the *delaunay* and *rgg* graphs is that the Kronecker graph is scale-free and thus utilizes the edge-parallel method for certain traversal iterations.

3.5 Related Work

Recent work on high performance graph algorithms has focused on accelerators, hybridization, and vectorization. Davidson *et al.* provide a GPU implementation to solve the Single-Source Shortest Path (SSSP) problem and also show a tradeoff between work-efficiency and available parallelism [34]. They compare the performance of various methods that all save work compared to the traditional Bellman-Ford approach. We consider the application of hybrid approaches such as the ones presented in this chapter to this problem to be an interesting direction of future work. Beamer *et al.* present a hybrid implementation of Breadth-First Search on multi-socket CPU server systems [9]. Similar to our hybrid approach, they use one approach ("top-down") when the vertex frontier is small and another ("bottom-up") when the vertex frontier is large. Their heuristic requires the size of the frontier, the number of edges to check from the frontier, and the number of edges to check from unexplored vertices. We alternatively use the change in the frontier size as the major factor in deciding how to distribute threads to units of work. Finally, Hong *et al.* provide a fast parallel detection of Strongly Connected Components in Small-World Graphs on multi-core CPUs [19]. Their work found limitations of previous approaches that were especially detrimental on large graph instances that exhibit the small-world phenomenon. The limitation of this approach is that it is not performance portable to general structures of network data, requiring users to have a priori knowledge of the topological structure of their input.

3.6 Conclusions

In this chapter we have discussed various methods for computing Betweenness Centrality on the GPU. Leveraging information about the structure of the graph, we present several methods that choose between two methods of parallelism: edge-parallel and work-efficient. For high-diameter graphs using asymptotically optimal algorithms is paramount

to obtaining good performance whereas for small-diameter graphs it is preferable to maximize memory throughput, even if unnecessary work is completed. In addition our methods are more scalable and general than existing implementations. Finally, we run our algorithm on a cluster of 192 GPUs, showing that speedup scales almost linearly with the number of GPUs. Overall, our single-GPU approaches perform 2.71× faster on average than the best previous GPU approach and our multi-GPU implementation is capable of exceeding 10 GTEPS.

For future work we would like to efficiently map additional graph analytics to parallel architectures. The importance of robust, high-performance primitives cannot be overstated for the implementation of more complicated parallel algorithms. Ideally, GPU kernels should be modular and reusable [56]; fortunately, packages such as Thrust [7] and CUB [8] are beginning to bridge this gap. A software environment in which users have access to a suite of high performance graph analytics on the GPU would allow for fast network analysis and serve as a building block for more complicated programs.

CHAPTER 4

STREAMING BETWEENNESS CENTRALITY ON THE GPU

The exploding popularity of online social networking has created a profound demand for high performance, scalable graph analytics. A particularly popular set of analytics attempt to measure *centrality*, or the importance of a given degree in its network. Such analyses can be used for contingency analysis for power grid component failures [24] or to find the best locations for stores within cities [37].

Architectural improvements haven't been fast enough to keep up with the demand for faster calculation of these analytics. In addition, many networks of practical interest are rapidly changing with time, exacerbating this issue. Hence, it is crucial for algorithms to be able to update analytics rather than recompute them. A lack of dynamic graph analytics in the literature leads to frameworks that perform static computations for graphs at different points in time. Repetitive static computations are wasteful since updates to the analytic typically only require computation on a small subset of the graph. The tremendous volume of updates to social networks and the web demands a high throughput solution that can process many updates in a given unit time. Thus the construction of dynamic graph analytics on the GPU is particularly useful for these applications. Although there has been recent work regarding irregular GPU computations [57], to our knowledge this chapter is the first attempt to implement a dynamic graph computation on the GPU.

The key contributions of this chapter are summarized below:

- We present several GPU implementations of dynamic betweenness centrality, the best of which can get significant speedup over 1) a CPU version of dynamic betweenness centrality and 2) the full recomputation of betweenness centrality on the GPU for a diverse set of graphs.

- We provide an analysis of the frequency of the various scenarios that can occur when

updating the graph and how each of these scenarios affect performance. Furthermore, we analyze the portion of the graph affected by each update and show that efficiently mapping threads to units of work is paramount to obtaining high performance.

- We analyze node and edge-based parallelism for our algorithm, noting that although edge-based parallelism has greater memory throughput, node-based parallelism has less contention over shared resources. Since the typical number of vertices to be processed at a particular instant is much less than the total number of edges in the graph, the node-based approach scales better and sees better performance characteristics overall.

The rest of the chapter is organized as follows. Section 4.1 focuses on a large amount of related work involving static and dynamic methods for calculating betweenness centrality. Section 4.2 presents our algorithms for dynamic betweenness centrality on the GPU and discusses performance optimization. Section 4.3 presents the benchmark graphs and target architecture used for this study. Section 4.4 presents our experimental results and analyses. Finally, Section 4.5 concludes and discusses future work.

4.1 Related Work

Betweenness centrality (BC) ranks the importance of a given vertex based on the number of shortest paths on which this vertex lies. Contemporary applications of betweenness centrality involve the study of AIDS within sexual networks [38], lethality in biological networks [58], community detection [23], and the analysis of the human brain [59]. The following subsections review various algorithms for computing static and dynamic betweenness centrality for large graphs.

4.1.1 Definitions

Given a graph $G = (V, E)$ with a set V of $n = |V|$ vertices and a set $E \subseteq V \times V$ of $m = |E|$ edges we define the following metrics. Let $d_s(t)$ be the distance of the shortest path from

the source vertex s to vertex t as is found by a Breadth-First Search (BFS). By definition $d_s(s) = 0$. Let σ_{st} represent the number of shortest paths starting at vertex s and ending at vertex t. Next, let $\sigma_{st}(v)$ denote the number of such shortest paths that pass through a particular vertex v. The betweenness centrality of a given vertex v can now be defined as:

$$BC(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{6}$$

Note that this metric is computationally intensive, as it requires the solution of the all-pairs shortest paths problem. Intuitively, the BC score of a vertex implies how often it is used as a connection on the shortest path of pairs of other vertices. Typically the vertices with the highest BC scores are of particular interest and the relative ranking of the vertices tends to be more informative than the magnitude of their scores [32].

4.1.2 Brandes's Algorithm

The fastest known sequential algorithm for computing betweenness centrality was presented by Brandes in 2001 [27]. He defines the *dependency* of a vertex v for a given source node s as:

$$\delta_s(v) = \sum_{w : v \in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w)) \tag{7}$$

In other words, the dependency of v is a function of the dependency of its immediate successors w. This recursive relationship allows for an efficient computation that avoids unnecessary additions from vertices that are not on shortest paths [60]. Using this definition, the BC score of a given vertex v can be computed as:

$$BC(v) = \sum_{s \neq v} \delta_s(v) \tag{8}$$

Algorithm 6 shows Brandes's approach to computing exact BC scores. For each node s in the graph, three stages are processed:

1. Initialization

2. Shortest Path Calculation

3. Dependency Accumulation

The first of these stages initializes local data structures. These data structures include a queue and stack for keeping progress during the later stages, a list of immediate predecessors for each element, the distance (d) of each element from the current root, the number of shortest paths (σ) from the root to each element, and the dependency (δ) of each element. The second stage is a Breadth-First Search (BFS) traversal that starts at the current root and finds the distance and the number of shortest paths from the current root to all other roots. Finally, the third stage visits nodes in the reverse order of the BFS traversal and finds the fraction of shortest paths that pass through each particular vertex out of all shortest paths. For undirected graphs, the algorithm has $O(mn)$ time complexity and $O(m+n)$ space complexity.

Since exact centrality computation on current workstations is infeasible for large-scale graphs, a number of methods for approximating BC scores have been proposed. One of these methods is to choose a subset of vertices to process in the outermost for loop (line 2) of Algorithm 6 [54]. Since the iterations of this for loop can all be processed in parallel, this approach scales well on multiple processors and is used to calculate BC scores for large graphs. If k nodes are chosen as *source nodes* (i.e. nodes to be processed in the outermost for loop of Algorithm 6) then the time complexity for approximating BC scores reduces to $O(mk)$. Since $k \leq n$, approximating the algorithm can be significantly faster than computing it exactly, depending on the values of k and n. For a detailed analysis regarding the accuracy of this approximation, we refer the reader to [54].

4.1.3 Parallel Implementations

Parallelism is another way to reduce the high computational cost of centrality metrics. Sariyüce et al. propose a heterogeneous implementation that extracts vertices of degree 1 from the graph, showing that only minor modifications to the calculation are necessary

Algorithm 6: Static Betweenness Centrality (Brandes) [27]

```
1  BC[v] ← 0, ∀v ∈ V
2  for s ∈ V do
3      Stage 1: Initialization
4      S ← empty stack; Q ← empty queue P[w] ← empty list, ∀w ∈ V
5      d[t] ← ∞, ∀t ∈ V
6      d[s] ← 0
7      σ[t] ← 0, ∀t ∈ V
8      σ[s] ← 1
9      δ[t] ← 0, ∀t ∈ V
10     Stage 2: Shortest Path Calculation
11     Q.enqueue(s)
12     while !Q.empty() do
13         v ← Q.dequeue()
14         S.push(v)
15         for w ∈ neighbors(v) do
16             //w found for the first time?
17             if d[w] = ∞ then
18                 Q.enqueue(w)
19                 d[w] ← d[v] + 1
20             //Shortest path to w via v?
21             if d[w] = d[v] + 1 then
22                 σ[w] ← σ[w] + σ[v]
23                 P[w].insert(v)
24     Stage 3: Dependency Accumulation
25     while !S.empty() do
26         w ← S.pop()
27         for v ∈ P[w] do
28             δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w])
29         if w ≠ s then
30             BC[w] ← BC[w] + δ[w]
```

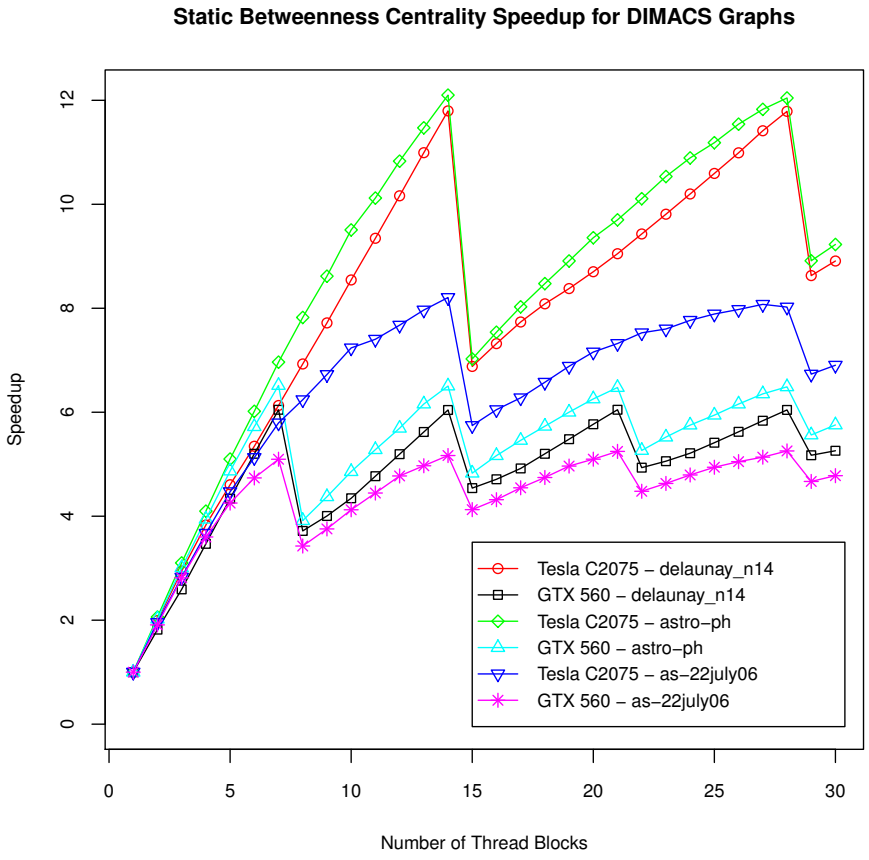


Figure 7: BC speedup relative to one thread block

after this removal [31]. Jia et al. propose a GPU implementation of betweenness centrality in [29]. The work of Jia et al. investigates the difference in performance between node and edge-based parallelism, concluding that edge-based parallelism gets better memory throughput and thus better performance. We revisit this comparison for our dynamic algorithms in Section 4.2. The optimal number of CUDA thread blocks was also investigated, but in less detail. The authors concluded that the optimal number of thread blocks is the number of Streaming Multiprocessors (SMs) on the GPU. Conventional wisdom with regard to GPU programming says that each SM should have multiple active thread blocks [61]; however, the claim from [29] seems to suggest that this strategy isn't ideal for irregular algorithms since the memory bus will become saturated.

To substantiate this claim and determine the best ratio of thread blocks to SMs we run

a static (and exact) betweenness centrality computation for a varying number of thread blocks and compare performance. We use three graphs from the DIMACS challenge as our input [47], using the largest graphs that are still feasible for an exact computation with contemporary hardware (i.e. graphs with tens of thousands of vertices). Figure 7 shows the speedup of static betweenness centrality relative to using one thread block for two GPUs: a GTX 560 with 7 SMs and a Tesla C2075 with 14 SMs. It is clear that the best performance is obtained by setting the number of thread blocks to be equal to the number of SMs or a multiple thereof, as concluded in [29]. For the graphs that we tested, we found that the performance of having one thread block per SM was slightly faster or about the same as the performance of having multiple thread blocks per SM. Hence we delegate one thread block per SM for the algorithms presented in the upcoming sections.

4.1.4 Dynamic Approaches

Three different algorithms for dynamic betweenness centrality have been proposed in recent literature. Lee et al. propose QUBE, an algorithm that updates BC scores by determining which vertices have BC values that may change, thus avoiding the full all-pairs shortest paths computation [62]. Kas et al. use a Java-based graph library to improve upon this result by directly updating the auxiliary data required by the algorithm [63]. Finally, Green et al. provide a high-performance implementation along with formal proofs and algorithms for the various scenarios that can occur when inserting or removing an edge [60]. Note that all of these approaches are sequential, making our implementation the first parallelized version of dynamic betweenness centrality. The implementation discussed in this chapter will most closely resemble the approach by Green et al. [60].

4.1.4.1 Update Scenarios

In this section the various scenarios for updating betweenness centrality scores are discussed in detail. Readers interested in formal proofs can find them in [60]. We restrict our focus to edge insertions since many real-world networks only experience growth and do

not shrink. For example, graphs resembling co-authorship will only expand as time progresses. Furthermore, it has been shown that edge removal updates require similar algorithmic techniques to edge insertion updates [62]. Thus, the lessons learned from focusing on edge insertions are directly applicable to edge deletions. It is also noteworthy that a node insertion causes no change to existing BC scores. A newly inserted node belongs to its own connected component (equivalently, has no incoming or outgoing edges) and thus has a BC score of 0. The new node will only affect the BC scores of other nodes once edges connect the new node to other connected components in the network.

To update the BC scores, we must store supplemental global data to the scores alone. For each source vertex s, the variables $d_s(t)$, σ_{st}, and $\delta_s(t)$ are preserved $\forall t \in V$. This added storage increases the space complexity to $O(n^2)$ for exact BC computation and $O(kn)$ for approximate BC computation using k source vertices; however, as we will show in Section 4.4 the performance gain is well worth the extra space.

Formally, an edge insertion $e = (u, v)$ creates a new graph $G' = (V, E')$ where $E' = E \cup \{e\}$. For each source vertex, one of the following three scenarios will occur, depending on the relation between u and v *before* the edge is inserted:

- Case 1: $|d_s(u) - d_s(v)| = 0$. The nodes connected to the inserted edge are the same distance from the source node. In terms of performance, this scenario is ideal because no additional work needs to be done for this source vertex (the other source vertices may require work, however). The reason that no additional work needs to be done is that the distances of u and v from the source do not change and no additional shortest paths are created. Note that this case can actually occur for two slightly different reasons: one when u, v, and s all belong to the same connected component and another when neither u nor v belongs to the same connected component as s.

- Case 2: $|d_s(u) - d_s(v)| = 1$. The nodes connected to the inserted edge are on adjacent levels. While none of the distances from the source vertex will change, it is possible

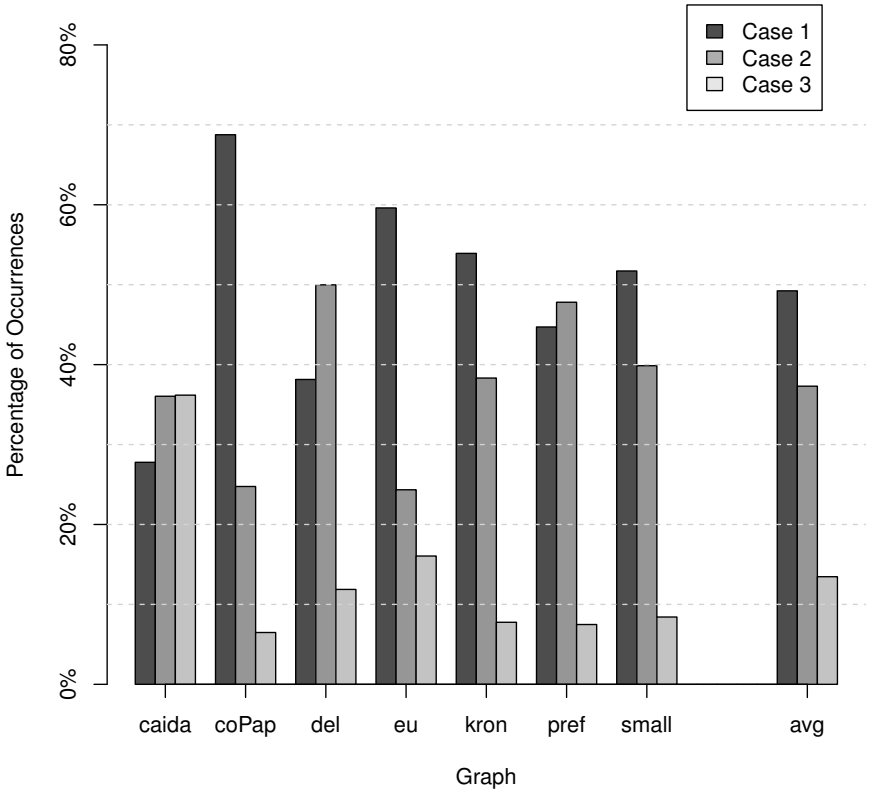


Figure 8: Distribution of scenarios for the graphs used in this study

that the number of shortest paths have changed and thus centrality scores may also change.

- Case 3: $|d_s(u) - d_s(v)| > 1$. The nodes connected to the inserted edge are greater than one level apart. In this case distances from the source vertex will change and shortest paths may have changed. Hence, centrality scores will need to be updated. Note that this case can actually occur for two slightly different reasons: one when u, v, and s all belong to the same connected component and another when either u or v (but not both) belong to the same connected component as s.

Figure 8 motivates the importance of the implemenation for Case 2 with regard to overall performance of the dynamic analytic. For each edge insertion, each source node will

face one of the three scenarios previously described. The data in Figure 8 reflects 100 edge insertions for each input graph. For each edge insertion, every source node in the graph faces one of the three scenarios described above. Therefore if k source nodes are used to approximate the BC scores of 100 edge insertions there will be $100k$ scenarios distributed among the 3 cases described above (up to $100n$ for the exact computation). Figure 8 shows how these distributions vary for the set of graphs used in this study. Recall from above that for Case 1, no work needs to be done. We can see that Case 2 represents 37.3% of all scenarios and 73.5% of the scenarios that require actual work (Cases 2 and 3) for this set of graphs. Hence, for the rest of this chapter we focus our analysis on Case 2, noting that our techniques generalize and can be applied to Case 3 and, oftentimes, parallel graph algorithms in general.

Algorithm 7 from Green et al. [60] shows how to update the intermediate variables and centrality scores for Case 2. In addition to d, σ, and δ, a few additional variables are introduced. Let t_v denote the stage of the update algorithm in which some vertex v was found. If $t_v = down$ then v was found in the shortest path (downward) calculation stage, if $t_v = up$ then v was found in the dependency accumulation stage, and if $t_v = untouched$ then v was not found in either stage. Also, let $\hat{\sigma}_{sv}$ and $\hat{\delta}_s(v)$ be the updated values of σ_{sv} and $\delta_s(v)$ after the insertion, respectively. Note that the algorithm takes the source node s as well as the endpoints u_{low} and u_{high} of the inserted edge. Since u_{low} and u_{high} belong to adjacent levels, one of them must be closer to s than the other. We refer to this closer node as being "higher" up in the BFS tree of s and hence call it u_{high}. Similarly, the other endpoint of the edge is "lower" in the BFS tree of s so we refer to it as u_{low}.

Lines 1 through 8 initialize these data structures. Note that a multi-level queue (QQ) is used in lieu of a stack because it is possible for nodes to be added to this "stack" in the dependency accumulation stage. The level order of the BFS tree from the source node s must be preserved as nodes are processed in the dependency accumulation stage. Processing in this stage begins with nodes that are the farthest away from s. If a node v at level i is

Algorithm 7: Dynamic Betweenness Centrality Case 2 (Green et al.) [60]

Input: Source node s and an inserted edge from u_{low} to u_{high}

Stage 1: Initialization
$Q \leftarrow$ empty queue
$QQ[level] \leftarrow$ empty queue, $level = 0, 1, \ldots, n-1$
$t[v] \leftarrow untouched, \forall v \in V \setminus \{u_{low}\}$; $t[u_{low}] \leftarrow down$
$\hat{\sigma}[v] \leftarrow \sigma[v], \forall v \in V \setminus \{u_{low}\}$; $\hat{\sigma}[u_{low}] \leftarrow \sigma[u_{low}] + \sigma[u_{high}]$
$\hat{\delta}[v] \leftarrow 0, \forall v \in V$
Stage 2: Shortest Path Calculation
$Q.enqueue(u_{low})$
$QQ[d[u_{low}]].enqueue(u_{low})$
while $!Q.empty()$ **do**
    $v \leftarrow Q.dequeue()$
    for $w \in neighbors(v)$ **do**
        if $d[w] = d[v] + 1$ **then**
            if $t[w] = untouched$ **then**
                $Q.enqueue(w)$
                $QQ[d[w]].enqueue(w)$
                $t[w] \leftarrow down$
            $\hat{\sigma}[w] \leftarrow \hat{\sigma}[w] + (\hat{\sigma}[v] - \sigma[v])$
Stage 3: Dependency Accumulation
while $level > 0$ **do**
    while $!QQ[level].empty()$ **do**
        $w \leftarrow QQ.dequeue()$
        for $v \in neighbors(w)$ **do**
            if $d[w] = d[v] + 1$ **then**
                if $t[v] = untouched$ **then**
                    $QQ[level-1].enqueue(v)$
                    $t[v] \leftarrow$ up
                    $\hat{\delta}[v] \leftarrow \delta[v]$
                $\hat{\delta}[v] \leftarrow \hat{\delta}[v] + \frac{\hat{\sigma}[v]}{\hat{\sigma}[w]}(1 + \hat{\delta}[w])$
                if $t[v] = up \wedge (v \neq u_{high} \vee w \neq u_{low})$ **then**
                    $\hat{\delta}[v] \leftarrow \hat{\delta}[v] - \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
                if $w \neq s$ **then**
                    $BC[w] \leftarrow BC[w] + \hat{\delta}[w] - \delta[w]$
    $level \leftarrow level - 1$
$\sigma[v] \leftarrow \hat{\sigma}[v], \forall v \in V$
for $v \in V$ **do**
    if $t[v] = untouched$ **then**
        $\delta[v] \leftarrow \hat{\delta}[v]$

pushed onto a stack rather than a multi-level queue in line 28 by a node w at level $i + 1$, the next node to be popped would be v instead of the remaining nodes at level $i + 1$ that have yet to be processed but must be processed first for correctness. Line 7 records the updated number of shortest paths for u_{low} due to the edge insertion. Since an edge is inserted from u_{high} to u_{low} all of the shortest paths that pass from s to u_{high} must also pass through u_{low} (because the new edge is the shortest path from u_{high} to u_{low}). Therefore $\hat{\sigma}[u_{low}]$ is initialized to $\sigma[u_{low}] + \sigma[u_{high}]$.

Lines 9 through 20 update the number of shortest paths from s to all other nodes due to the insertion of the new edge. Since we know that the number of shortest paths for nodes between s and u_{high} will not change, we can start the BFS traversal downward from u_{low}. Note that this approach does not explicitly store predecessors as is done in line 23 of Algorithm 6. Instead, the dependency accumulation stage looks at all neighbors of nodes popped from the multi-level queue and checks to see that a given neighbor is a predecessor before subsequent processing (line 26). Although this method generates some additional work it has been shown to save $O(E)$ memory in addition to showing speedups in practice [42].

Finally, lines 21 through 40 update the BC scores of each node. Since the preceding stage potentially changed the number of shortest paths from the root s to other nodes and since the dependency is a function of the number of shortest paths, the values of the dependency will potentially change as well. Line 31 adds the correct contribution of w to the dependency of it's predecessor v and line 33 subtracts out the prior contribution of w to the dependency of v, which is now incorrect due to the edge insertion. Lines 37 through 40 copy the updated (local) values of shortest paths and dependency to global variables to be used for the next update.

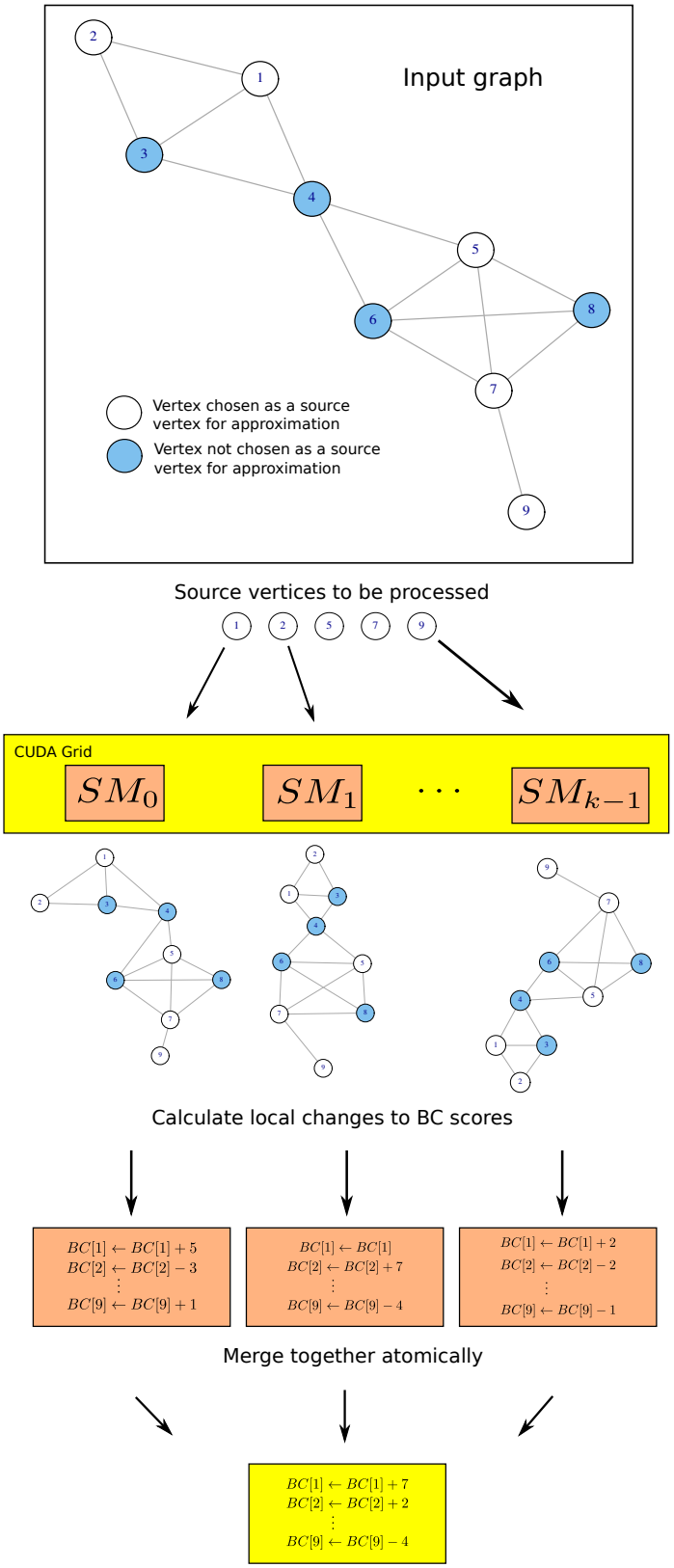


Figure 9: Decomposition of work to parallel compute units

4.2 Dynamic Betweenness Centrality on the GPU

In this section we present several of our GPU implementations for dynamic betweenness centrality computations. Since figuring out which case each source node has to compute is trivial, we focus on the algorithmic challenges of the cases themselves. Again, our discussion focuses on Case 2 (edge insertion between nodes on adjacent levels) due to the motivation from Figure 8 and its discussion in the previous section.

Similar to previous work [29], we assign the maximum number of threads per block and set the number of thread blocks to be equal to the number of streaming multiprocessors for all kernels. Each thread block takes advantage of the available coarse-grained parallelism by handling independent source vertices while the threads within a block take advantage of fine-grained parallelism by traversing graph edges and updating state concurrently.

Algorithm 8: Kernel for initialization of local variables

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

```
1 Stage 1: Initialization
2 for v ∈ V do in parallel
3     if v = u_low then
4         t[v] ← down
5         σ̂[v] ← σ[v] + σ[u_high]
6     else
7         t[v] ← untouched
8         σ̂[v] ← σ[v]
9     δ̂[v] ← 0
```

Figure 9 illustrates this concept. To approximate BC, a subset of the graph's vertices are chosen at random and used as root nodes for shortest path calculations (shown as the unfilled vertices of the input graph in Figure 9). Each Streaming Multiprocessor (SM) takes one source vertex and performs a BFS to calculate the number of shortest paths from that vertex to all other vertices in the graph. These shortest path calculations are independent among SMs and can hence be performed in parallel without communication overhead. The dependency accumulation stage is also independent among SMs with the exception of the

final update to the BC value itself. To update the global array holding the BC scores each SM adds its changes atomically, preventing data races. Since GPUs currently tend to have a small number of SMs (< 50) and since these additions are not necessarily performed concurrently (because one SM can finish its updates independently of the others), there is little contention for global memory resources for these atomic additions. Thus the use of atomic operations in this instance is admissible as it has negligible overhead.

Throughout this section we compare two approaches of fine-grained parallelism: *edge-based* and *node-based*. *Edge-based* parallelism assigns one thread to each edge in the graph, which results with a greater number of work units that consist of a small, roughly equivalent amount of work. *Node-based* parallelism, on the other hand, assigns one thread to each vertex in the graph, which results in fewer work units that have varying size. Note that both methods use the same number of threads, but map threads to work differently. Since there are typically more vertices and edges in a graph than available threads, each thread will process multiple units of work. For example, if there are 1000 available threads and 4000 edges in the graph, the edge-based method will provide each thread with 4 edges to process. The edge-based approach has better load balancing and has been shown to generate greater memory throughput for static betweenness centrality on the GPU [29] whereas the node-based approach has less contention over shared resources. Both of these approaches initialize local variables in parallel in the same way, as shown in Algorithm 8.

4.2.1 Updating the Number of Shortest Paths

Algorithm 9 shows GPU pseudocode to update the number of shortest paths from the source node s to all other nodes in the graph using edge-based parallelism. The **shared** keyword is used to denote variables that are explicitly stored in the GPU's fast scratchpad (or shared) memory. Threads within an SM will see the same value of shared variables while threads belonging to different SMs will not. Note that explicit queues aren't necessary as shared memory and synchronization are used to ensure that vertex frontiers (depths) are processed in the correct order. It is possible for multiple threads to successfully execute Line 10 and

Algorithm 9: Edge-based Parallel Shortest Path Calculation Kernel

```
   Input: Source node s and endpoints u_low and u_high of the inserted edge
1  Stage 2: Shortest Path Calculation
2  shared current_depth ← d[u_low]
3  shared done ← false
4  while !done do
5      done ← true
6      for (v, w) ∈ E do in parallel
7          if d[v] = current_depth then
8              if d[w] = current_depth + 1 then
9                  if t[w] = untouched then
10                     t[w] ← down
11                     done ← false
12                 atomicAdd(&σ̂[w], σ̂[v] − σ[v])
13     barrier()
14     current_depth ← current_depth + 1
```

set $t[w]$ to *down*, leading to a data race; however, this data race is considered benign as it has no effect on program output. Line 12 requires an atomic (serialized) write to $\hat{\sigma}[w]$ to prevent a data race. Otherwise, this calculation is exactly the same as the one in Line 20 of Algorithm 7.

Alternatively, Algorithm 10 shows GPU pseudocode that achieves the same result using node-based parallelism. We introduce three different arrays that act as queues in lines 2-6. The Q array holds nodes that are being processed in the current level of the BFS traversal. The $Q2$ array is used to hold vertices found in the current level of the BFS traversal. These vertices are transferred to Q (line 26) and are explored in the next level of the BFS traversal. Separate queues are necessary because all nodes at the current level must be processed before any nodes at the following level are to be processed to ensure correctness. Finally, the QQ array holds nodes that are to be processed during the dependency accumulation, analagous to the multi-level queue used in Algorithm 7.

Using this approach, the number of threads needed to process an iteration is simply the number of nodes that currently reside in Q (which is stored in the variable Q_{len}). In

Algorithm 10: Node-based parallel Shortest Path Calculation Kernel

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

```
1  Stage 2: Shortest Path Calculation
2  shared current_depth ← d[u_low]
3  Q[0] ← u_low
4  Q_len ← 1
5  Q2_len ← 0
6  QQ[0] ← u_low
7  QQ_len ← 1
8  while true do
9      for tid ← 0 ... Q_len − 1 do in parallel
10         v ← Q[tid]
11         for w ∈ neighbors(v) do
12             if d[w] = d[v] + 1 then
13                 if t[w] = untouched then
14                     t[w] ← down
15                     i ← atomicAdd(&Q2_len, 1)
16                     Q2[i] ← w
17                 atomicAdd(&σ̂[w], σ̂[v] − σ[v])
18     barrier()
19     if Q2_len = 0 then
20         break
21     else
22         remove_duplicates(Q2, Q2_len)
23         Q_len ← Q2_len
24         Q2_len ← 0
25         for tid ← 0 ... Q_len − 1 do in parallel
26             Q[tid] ← Q2[tid]
27             i ← atomicAdd(&QQ_len, 1)
28             QQ[i] ← Q2[tid]
29     barrier()
30 for v ∈ V do
31     atomicMax(&current_depth, d[v])
```

contrast, the edge-based parallel approach spawns $|E|$ threads for every level of the search, regardless of the amount of work needed to be done. Hence the edge-based approach, despite being conceptually simpler and more convenient to program, generates significantly more accesses to memory, most of which are futile.

It is important to recognize that $Q2$ may have duplicate entries whereas Q and QQ will not. An atomic operation could be used to test and set $t[w]$ on line 13, ensuring that only one thread places w into $Q2$ on line 16. We avoid this atomic operation by allowing multiple threads to insert the same node into $Q2$ and removing duplicate entries from $Q2$ (line 22) before transferring $Q2$ to Q for the next iteration of the search. Note that we pass $Q2_{len}$ to the *remove_duplicates*() subroutine because the removal of duplicates reduces the size of the queue. Duplicate entries are removed from $Q2$ by the following procedure (similar to Merrill et. al [10]):

1. Sort the elements of $Q2$. In our implementation we use bitonic sort, though we consider this choice to have a negligible impact on performance because $Q2_{len}$ is typically much smaller than n.

2. Compare the value at index $i - 1$ from the value at index i of the sorted array. Using an additional array, mark index i with the value *true* if the compared values are equivalent. Else, mark *false*. This output represents which indices of $Q2$ correspond to unique elements.

3. Perform a prefix sum on the above result to determine which indices into Q each corresponding unique element of $Q2$ should be placed and to find the number of unique entries in the queue (i.e. Q_{len} for the next search iteration).

After the above procedure, the unique entries in $Q2$ are transferred to Q for the next BFS iteration (line 26). These entries are also added to QQ (line 28) for the dependency accumulation stage. Lines 30 and 31 set the appropriate distance of the furthest processed

vertex from s as the starting point of the dependency accumulation method discussed in the next section.

Algorithm 11: Edge-based Parallel Dependency Accumulation Kernel

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

```
1  Stage 3: Dependency Accumulation
2  while current_depth > 1 do
3      for (v, w) ∈ E do in parallel
4          if d[v] = current_depth then
5              if d[w] = current_depth − 1 then
6                  dsv ← 0
7                  if atomicCAS(&t[v], untouched, up) = untouched then
8                      dsv ← dsv + δ[v]
9                  dsv ← dsv + (σ̂[v] / σ̂[w]) * (1 + δ̂[w])
10                 if t[v] = up ∧ (v ≠ u_high ∨ w ≠ u_low) then
11                     dsv ← dsv − (σ[v] / σ[w]) * (1 + δ[w])
12                 atomicAdd(&δ̂[v], dsv)
13     barrier()
14     current_depth ← current_depth − 1
```

4.2.2 Updating the Dependency Accumulation

Once the shortest path calculation has been updated, it remains to update the dependencies and the BC scores themselves. Algorithm 11 shows an edge-based parallel implementation of the dependency accumulation. Continuing from where Algorithm 9 left off, vertices of decreasing distance from the source are processed one level at a time. Line 7 requires an atomic operation that ensures that only the first thread to attempt to successfully set $t_v = up$ executes Line 8. The *atomicCAS*() function does an atomic compare and swap. If $t[v] = untouched$, the function sets $t[v] = up$ and returns *untouched*. Otherwise, the function doesn't change the contents of $t[v]$ and returns the value of $t[v]$ provided to the function, causing the if statement on Line 7 to evaluate to $false$ so that Line 8 will not execute. The register dsv is used to accumulate all changes to $\hat{\delta}[v]$ brought upon by w so that only one atomic addition (Line 12) to update $\hat{\delta}[v]$ is necessary.

Algorithm 12: Node-based parallel Dependency Accumulation Kernel

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

Stage 3: Dependency Accumulation
while $current\_depth > 1$ **do**
  for $tid \leftarrow 0 \dots QQ_{len} - 1$ **do in parallel**
    $w \leftarrow QQ[tid]$
    if $d[w] = current\_depth$ **then**
      for $v \in neighbors(w)$ **do**
        if $d[v] = current\_depth - 1$ **then**
          $dsv \leftarrow 0$
          if $atomicCAS(\&t[v], untouched, up) = untouched$ **then**
            $dsv \leftarrow dsv + \delta[v]$
            $i \leftarrow atomicAdd(\&Q2_{len}, 1)$
            $QQ[i + QQ_{len}] = v$
          $dsv \leftarrow dsv + \frac{\hat{\sigma}[v]}{\hat{\sigma}[w]} * (1 + \hat{\delta}[w])$
          if $t[v] = up \wedge (v \neq u_{high} \vee w \neq u_{low})$ **then**
            $dsv \leftarrow dsv - \frac{\sigma[v]}{\sigma[w]} * (1 + \delta[w])$
          $atomicAdd(\&\hat{\delta}[v], dsv)$
  $barrier()$
  $QQ_{len} \leftarrow QQ_{len} + Q2_{len}$
  $Q2_{len} \leftarrow 0$
  $current\_depth \leftarrow current\_depth - 1$

The corresponding node-based parallel dependency accumulation is shown in Algorithm 12. To simulate the multi-level queue seen in Algorithm 7 we place processed vertices from all levels of the BFS traversal into one array (QQ), as shown in Algorithm 10. To process this array in level synchronous order, we have threads extract vertices from the array and check if the level of the extracted vertices matches the current level that is to be processed, as shown on line 5. If we find a node that wasn't touched in the shortest path calculation stage we can safely add it to the end of QQ (line 12) and safely process it concurrently with other nodes at its level because of this check. Again, since only QQ_{len} threads are performing work whereas $|E|$ threads are performing work in the edge-based approach, the node-based approach exhibits significantly less memory traffic. Since QQ_{len} is the number of nodes to be processed at all levels and not just the current level, even the node-based approach performs some unnecessary work. However, we will show in Section 4.4 that the amount of this extra work is tremendously small in virtually all cases.

Algorithm 13: Kernel to update global variables

```
   Input: Source node s and endpoints u_low and u_high of the inserted edge
1  for v ∈ V do in parallel
2      if v ≠ s ∧ t[v] ≠ untouched then
3          atomicAdd(&BC[v], δ̂[v] − δ[v])
4      σ[v] ← σ̂[v]
5      if t[v] ≠ untouched then
6          δ[v] ← δ̂[v]
```

Once the dependency accumulation kernel is complete the updated values of the dependency are used to adjust the BC scores and the local variables $\hat{\sigma}$ and $\hat{\delta}$ are copied to their respective global variables σ and δ for the next update. Algorithm 13 shows how to perform this task in parallel. Similar to the initialization in Algorithm 8, both the edge and node-based approaches complete this task in the same way.

Table 5: Suite of benchmark graphs

Name	Vertices	Edges	Significance
caidaRouterLevel (caida)	192,244	609,066	Internet Router Level Graph
coPapersCiteseer (coPap)	434,102	16,036,720	Social Network
delaunay_n20 (del)	1,048,576	3,145,686	Random Triangulation
eu-2005 (eu)	862,664	16,138,468	Web Crawl
kron_g500-simple-logn19 (kron)	524,288	21,780,787	Kronecker Graph
preferentialAttachment (pref)	100,000	499,985	Scale-free [64]
smallworld (small)	100,000	499,998	Logarithmic Diameter [40]

4.3 Experimental Setup

Table 5 shows the graph inputs used throughout this study. Again, we focus on approximate calculation of betweenness centrality since we are interested in the analysis of large graphs. The graph data was downloaded from the 10th DIMACS challenge [47]. These graphs were chosen based on size, diversity, and relevance to dynamic graph analytics. The set of graphs consists of real-world and random graphs and different classes of graphs are represented, such as small-world and scale-free graphs.

Single-threaded CPU experiments are implemented in C++ and compiled with `gcc -O3 -std=c++0x` flags and GPU experiments are implemented using CUDA and compiled with `nvcc -O3 -arch=sm_21` flags. The CPU used in this study is an Intel Core i7-2600K Processor running at 3.4GHz with an 8MB cache and 16GB of DRAM. The GPU used in this study is an Nvidia Tesla C2075 with 14 streaming multiprocessors each consisting of 32 stream processors that run at 1.15 GHz. The Tesla C2075 has 6GB of GDDR5 memory and has compute capability 2.0.

For each dynamic computation, 100 edges are chosen at random to be removed from the graph, similar to the approaches used in [62] and [63]. These edges are then reinserted into the graph one at a time and the analytic is updated. We choose $k = 256$ source nodes for approximation of BC, also at random, following the guidelines of the DARPA Scalable Synthetic Compact Applications (SSCA) benchmark suite [65]. To ensure that the proposed experiments are fair, the BC scores are approximated by all implementations: the dynamic

Table 6: Comparison of Dynamic CPU and Dynamic GPU Algorithms

Graph	CPU Time (s)	Method	GPU Time (s)	Speedup
caida	1749.98	Edge	84.79	20.64x
		Node	**15.85**	**110.41x**
coPap	1080.81	Edge	762.81	1.41x
		Node	**20.49**	**52.75x**
del	4762.75	Edge	4611.52	1.03x
		Node	**196.48**	**24.24x**
eu	3991.27	Edge	591.20	6.75x
		Node	**71.23**	**56.03x**
kron	1951.86	Edge	1668.27	1.17x
		Node	**81.54**	**23.94x**
pref	380.77	Edge	62.73	6.07x
		Node	**10.38**	**36.68x**
small	360.82	Edge	29.14	12.38x
		Node	**7.20**	**50.11x**

CPU baseline from Green et al. [60], our dynamic node and edge parallel GPU algorithms, and the static BC computation on the GPU from Jia et al. [29]. For each experiment we compare the results of the baseline and our algorithms to ensure that both yield the same results. We neglect the cost of updating the graph, choosing to focus on the design and performance of the analytic itself. Several techniques for dynamically updating graph data structures at a small amortized cost are disccused in [66].

4.4 Experimental Results

Table 6 shows the speedup of our dynamic GPU BC implementations over the dynamic sequential CPU algorithm from Green et al. [60] for both the edge and node-based parallel methods. We can see that although the edge-based parallel method can significantly outperform the CPU in some cases, the node parallel method substantially improves upon this result. It is clear that the edge-based approach does not scale well to larger graphs because the amount of unnecessary work that it performs grows with the size of the graph. Since the edge-based approach assigns one thread for every edge in the graph and since only a small subset of edges need to be traversed for a specified iteration, the edge-based

approach ends up with many threads that perform an unnecessary comparison for a branch instruction along with the loads it depends on. In contrast, the node-based method assigns one thread for every element in the queue being processed. In the shortest path calculation stage each of these elements has necessary work to complete, which means that this thread mapping is perfectly work efficient. In the dependency accumulation stage only a subset of these elements have necessary work to complete although the size of the queue is $O(n)$ (and in practice typically much smaller than n), which is significantly less than the number of edges in the graph, particularly for sparse graphs. Hence the node-based approach still provides a notably better mapping of threads to units of work. The node-based method performs well even for scale-free graphs such as *preferentialAttachment* with power-law degree distributions that can lead to severe workload imbalance among threads. We can see that our node-parallel GPU approach is up to 110x faster than the sequential CPU approach for the set of graphs used in this study.

In addition to providing speedups over a single-threaded CPU implementation of the dynamic algorithm, our approach also provides high performance in comparison to a full recomputation of the analytic on the GPU. Table 7 compares the execution time of a static BC computation using the implementation available from [29] to the slowest, average, and fastest updates from our optimized node-parallel dynamic algorithm. We can see that, even in the worst case for each graph a dynamic update is faster than a static recomputation. Intuitively this result makes sense because the number of edges traversed by the static computation is an upper bound for the number of edges that need to be traversed by the dynamic computation.

The fastest updates occur when all source nodes see a Case 1 scenario. Since the Case 1 scenario requires no work, if all source nodes see this scenario then no source nodes require work and the edge insertion has no effect on BC scores. This ideal scenario took place for one or more of the edge insertions for *caidaRouterLevel*, *coPapersCiteseer*, *delaunay_n20*, and *eu-2005*. We can see from Table 7 that these updates all took ~ 0.0003 seconds, which

Table 7: Comparison of Node Parallel GPU Updates to GPU Recomputation

Graph	**Recomputation (s)**	**Update (s)**	**Speedup**
caida	1.99	Slowest: 0.3295	6.05x
		Average: 0.1585	12.58x
		Fastest: 0.0003	6095.09x
coPap	31.35	Slowest: 0.7242	43.31x
		Average: 0.2049	153.02x
		Fastest: 0.0003	94729.29x
del	99.60	Slowest: 10.8997	9.14x
		Average: 1.9648	50.69x
		Fastest: 0.0003	296436.91x
eu	21.40	Slowest: 3.0308	7.06x
		Average: 0.7123	30.04x
		Fastest: 0.0003	64445.53x
kron	38.69	Slowest: 1.5658	24.71x
		Average: 0.8154	47.45x
		Fastest: 0.2725	141.96x
pref	1.27	Slowest: 0.5907	2.15x
		Average: 0.1038	12.24x
		Fastest: 0.0603	21.07x
small	0.68	Slowest: 0.0978	6.98x
		Average: 0.0720	9.48x
		Fastest: 0.0350	19.49x

is simply the amount of time necessary to discover that none of the BC scores will change due to the insertion. The speedups seen for this ideal case are essentially bounded by how long a recomputation takes, which is heavily dependent on the size of the graph. In contrast, the fastest cases for *kron_g500-simple-logn19*, *preferentialAttachment*, and *small-world* require updates to BC scores from one or more of the source nodes. For example, the fastest edge insertion for *kron_g500-simple-logn19* led to a Case 1 scenario for 222 of the 256 source nodes used for approximation, which means that the remaining 34 source nodes required significant amounts of work. Hence, we see the large difference between the fastest cases between the various input graphs.

More typically, many or even all source nodes can require significant work. This result is more likely to occur in practice because it is unlikely for two nodes to be on the same level of a breadth-first search from the perspective of every other node in the graph. However, even the slowest graph updates are preferable to a full recomputation of the BC scores, with speedups ranging from 2x to 43x for the graphs used in this study. The key takeaway is that the updates can ignore the portion of the graph between the root node and u_{high} when counting the number of shortest paths as the insertion of an edge below these vertices cannot create new shortest paths from the root to these vertices. Depending on the distance from the root node to the nodes connecting the inserted edge, the number of shortest path computations that can be neglected in this way can be very significant. Hence, the amount of time that a given update takes in not solely dependent on how many of the source nodes require work; it is also dependent on *how much* work each of those source nodes requires. The scatterplot in Figure 10 illustrates this concept. For each occurrence of a Case 2 scenario in each graph we record the number of nodes that are "touched" (i.e. $\{i \in V \mid t[i] \neq untouched\}$). We divide these counts by the total number of nodes in the graph and sort them from least to greatest to see the distribution.

The results shown in Figure 10 are quite surprising. Of the 62,844 Case 2 scenarios encountered across the graphs used in this study, the largest percentage of nodes that were

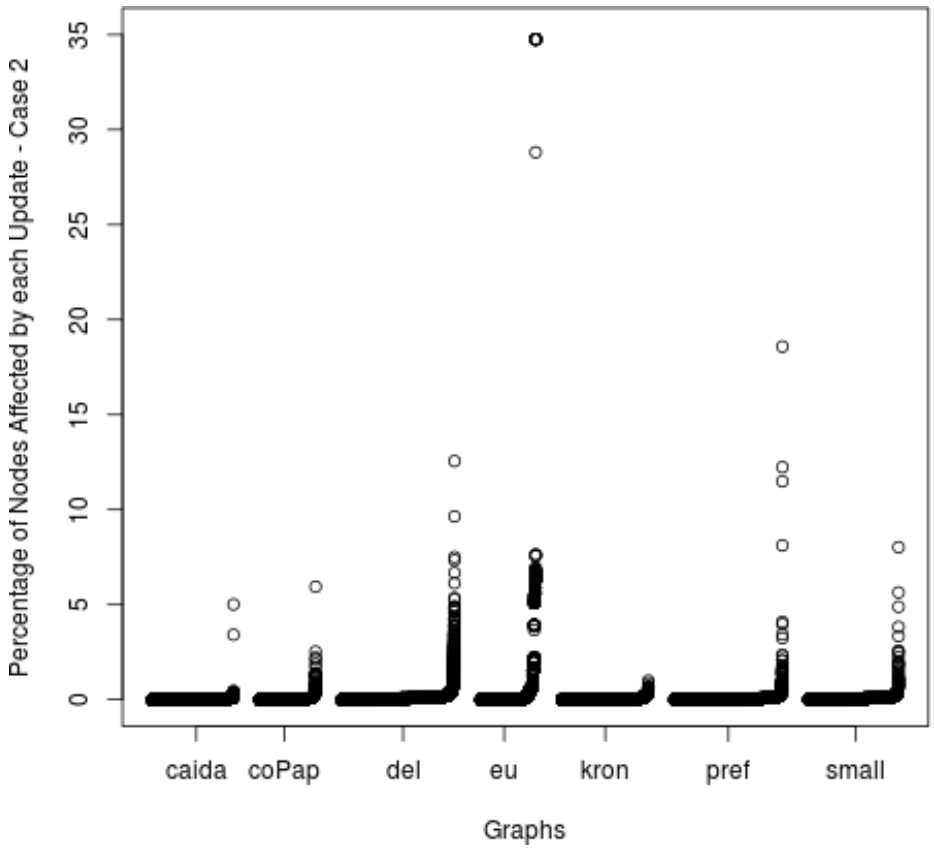


Figure 10: Portion of the graph that is touched for each Case 2 scenario

touched was approximately 35%. Perhaps even more surprising is that a vast majority of the Case 2 scenarios touched an extremely small portion of the graph, as evidence by the dense collection of points toward the bottom of the figure. These results strongly motivate the need for proper mapping of threads to units of work in high performance algorithm design for dynamic graph analytics. Through a combination of being able to detect when updates to the graph will not affect BC scores and avoiding unnecessary accesses to memory and redundant computation when graph updates do affect BC scores we achieve an average of a 45x speedup over a full recomputation of BC scores on the GPU across all graphs used in this study.

4.5 Conclusions

In this chapter we compared two GPU implementations of dynamic betweenness centrality, leveraging analyses from a wealth of related work. To our knowledge, this is the first time GPUs have been used for the analysis of time-varying graphs. By comparing these two different decompositions of threads to units of work, we show that keeping explicit track of the work that needs to be done is a vastly superior strategy. Although our approach uses atomic operations that serialize certain accesses to memory, we show that these memory locations are typically in low contention among threads because a surprisingly small number of nodes are affected by each update. Our approach achieves up to a 110x speedup over a CPU implementation of the algorithm and can update the analytic 45x faster on average than a static recomputation of the analytic on the GPU.

A multitude of opportunities exist to extend this area of work. If a linear space dynamic betweenness centrality algorithm exists, it would allow scaling of the techniques in this chapter to significantly larger graphs. Further performance improvements can be attained with multi-GPU, heterogeneous, or distributed implementations of this algorithm. The vast amount of coarse-grained parallelism that exists should allow for excellent strong scaling for such implementations. Finally, there are plenty of other graph algorithms that can benefit from either dynamic implementations or parallelism on multi-core CPUs and GPUs.

CHAPTER 5

OPTIMIZING TIME AND ENERGY FOR GPU BETWEENNESS CENTRALITY

Graphs are used to model the structure of the internet [67], interactions in social communties [68], and dynamic simulations of physical phenomena [69]. Many common graph problems have efficient sequential solutions but resist attempts at parallel efficiency. Increasingly parallel architectures and accelerators require new algorithms for both performance and power efficiency. The high memory bandwidth and power efficiency of Graphics Processing Units (GPUs) make them attractive to bandwidth-hungry graph algorithms, but mapping the analytics to GPU hardware is challenging.

Graph analysis algorithms often require fine-grained synchronization that limits parallelization. Some algorithms, like lexicographic depth-first search, are known to be P-complete and are inherently sequential [70]. Limited spatial locality and widely varying computational load also present challenges beyond those common in scientific computing or map-reduce-style data analysis. Maintaining analytics as new data streams into the graph without entirely recomputing results is another new challenge.

This chapter tackles these challenges with the following contributions:

- We propose various parallel methods for calculating betweenness centrality (BC), a successful analytic that tracks the influence of vertices in a network. We consider both coarse-grained and fine-grained methods of parallelism.

- We compare static methods for re-computing BC scores to a natively dynamic method that updates BC scores. We show that most edge changes affect a surprisingly small portion of the graph and that asymptotically efficient algorithms are crucial to analyzing time-varying graphs.

- We present results comparing our methods to the state-of-the-art on embedded and

HPC platforms considering both time and energy to solution. Our static implementation of the algorithm is capable of exceeding 2 MTEPS/W. Our dynamic implementation of the algorithm achieves greater than a 25× speedup over existing sequential methods on the CPU. On the GPU, our implementation achieves on average a 6.9× speedup and 83% reduction in energy consumption compared to a static recomputation.

5.1 Background

5.1.1 GPU Computing

Although GPUs are typically known for rendering computer graphics, the introduction of programming models such as CUDA and OpenCL have opened the computational power of the GPU to domains such as databases, electronic design automation, and biology [24, 71, 72]. GPUs have been successful in accelerating compute-bound applications that have regular structure and lots of floating point arithmetic [73]. Recent research also has shown successful acceleration of irregular and memory-bound applications that have randomized memory access patterns [10, 34].

GPUs are designed for highly parallel operation and dedicate transistors to arithmetic units rather than branch predictors or large caches. They leverage a single-instruction, multiple-thread (SIMT) programming model where consecutive threads execute the same instruction on different elements of data. A GPU consists of a number of streaming multiprocessors (SMs) that each execute threads in groups, known as warps on NVIDIA's GPUs. In the case of a branch instruction, the resulting paths of the branch are executed sequentially by predicated execution.

Programmers using NVIDIA's CUDA specify a number of *grid* and *block* dimensions for each kernel. These dimensions specify how many groups of threads are assigned to each SM and how many threads coexist within those groups. Programmers also manage *shared memory*, which is scratchpad storage assigned to each SM. Shared memory has much higher bandwidth than global memory but is smaller and hence harder to use in

applications that require data scalability.

Compared to conventional CPUs, GPUs tend to consume more instantaneous power but provide significantly higher throughput which results in better overall energy efficency in terms of performance per Watt. For instance, all of the top 10 computers on the November 2013 Green500 list utilize GPU accelerators [74].

5.1.2 Betweenness Centrality

Centrality metrics are an important class of graph algorithms used in applications such as graph visualization [75], urban planning [76], and community detection [77]. Betweenness Centrality (BC) was a metric developed in the social sciences for tracking the control of information in communication networks [22]. Recently it has been used to determine influential members of social networks [36]. BC scores are obtained by calculating the ratio of the number of times a vertex is on a shortest path between pairs of other vertices to the total number of shortest paths between those vertices.

Let σ_{st} be the number of shortest paths between vertices s and t and let $\sigma_{st}(v)$ be the number of these paths that pass through a particular vertex v. The betweenness centrality of v can be defined in terms of these numbers as follows:

$$BC(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{9}$$

The fastest known sequential algorithm for computing BC scores was developed by Brandes [27]. Rather than using the $O(n^3)$ Floyd-Warshall algorithm to solve the all-pairs shortest path (APSP) problem, Brandes derived a recursive relationship between vertices and their successors. The algorithm performs a breadth-first search traversal to solve the APSP problem and uses these results in a backward traversal on the graph referred to as the dependency accumulation to recursively obtain the centrality scores. Even with these improvements, the algorithm is computationally demanding as it requires $O(mn)$ time for unweighted graphs, where n is the number of vertices and m is the number of edges in the graph.

Several high-level strategies have been used to accelerate the computation of betweenness centrailty, such as approximation techniques [53], parallelism [32], and streaming [2]. The simplest method of approximating BC scores is to use a subset of the source vertices for the calculation. This step reduces the time complexity of the algorithm from $O(mn)$ to $O(mk)$ where k is the number of approximated source vertices. Essentially, the number of shortest paths from the k vertices to all vertices are found instead of the number of shortest paths between all pairs of vertices. Betweenness centrality lends itself well to parallelism since both coarse and fine-grained opportunities for parallelizing Brandes's algorithm exist. Coarse-grained parallelism involves assigning different source vertices to different threads or compute units. This assignment of work is embarassingly parallel since all source vertices can be handled independently. Fine-grained parallelism of BC assigns threads to cooperatively execute stages of the graph traversal needed for the shortest path calculation and dependency accumulation stages. Finally, several methods for incrementally updating centrality scores rather than recomputing them have been proposed in the literature [60]. Streaming methods are becoming increasingly important to analyze *dynamic* graphs that change over time. Typical network updates only affect a local region of the graph, making global recomputations wasteful in terms of both time and energy. Experimental results for both our static and dynamic implementations of Betweenness Centrality on the GPU can be found in Section 5.4.

5.2 Methodology

5.2.1 Coarse-grained Parallelism

The most important consideration for both our static and dynamic implementations of betweenness centrality is the decomposition of threads to units of work. Previous work investigating absolute performance showed that the number of thread blocks should be equivalent to the number of SMs for calculating betweenness centrality [29]. This also proves true for energy efficiency. Figure 11 shows how the average instantaneous power consumption of a Tesla C2075 GPU varies with thread blocks. Since the C2075 has 14 SMs, we can see

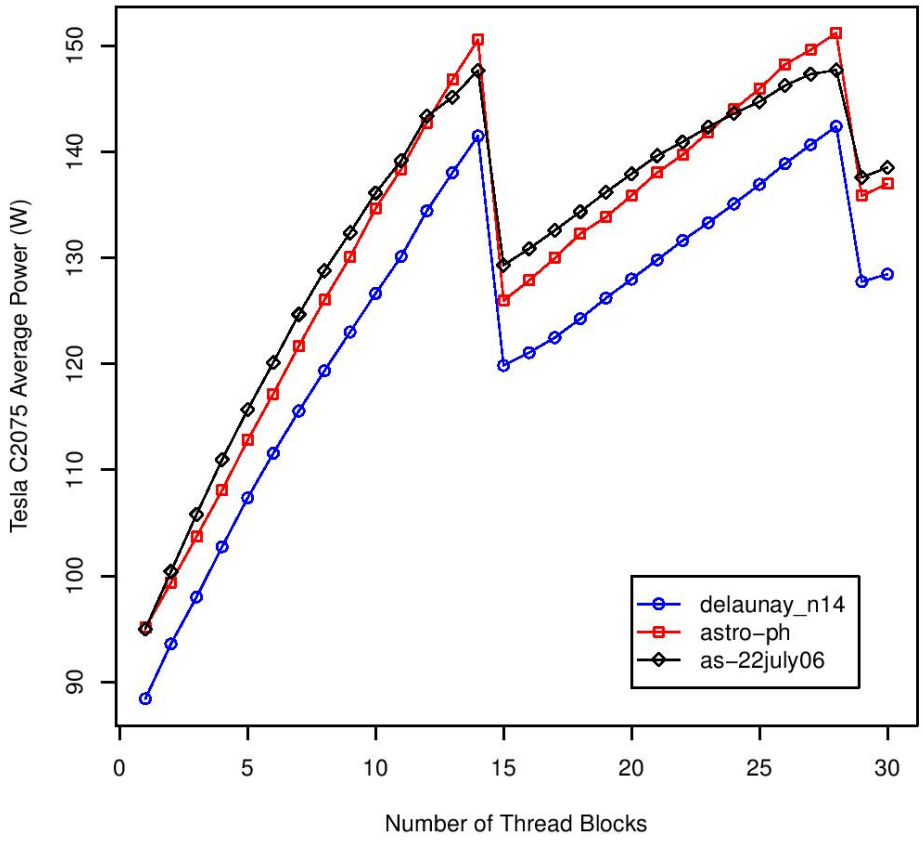


Figure 11: Power consumed as a function of the number of thread blocks launched

that the most power is consumed when the number of thread blocks issued is a multiple of the number of SMs. Interestingly, it appears that when 15 blocks are issued, rather than scheduling one block to each SM with one block leftover, the hardware opts to issue two blocks to 7 of the SMs and one block to an 8th SM in an attempt to conserve power by idling the remaining 6 SMs. Noting the scale of the y-axis, assigning thread blocks to all of the SMs on the GPU requires less than twice as much power than using just one thread block. Since the performance of the algorithm scales linearly with the number of active SMs (because each SM can execute independently in parallel), assigning one thread block to each SM is clearly the most energy-efficient method of operation.

5.2.2 Fine-grained Parallelism

Each cooperative thread array (CTA), or thread block, of the GPU is assigned a root vertex to traverse from and perform shortest path and dependency calculations. This results in attributing that root vertex's impact on the BC scores. Once this process has been completed for all of the roots in the graph (or all of the roots to be approximated), the algorithm terminates. The threads within each CTA work together to traverse the graph and calculate shortest paths and dependencies in parallel. Figure 12 illustrates this process. Each root, or source vertex, to be processed is assigned to a CTA that is scheduled to one of the SMs on the GPU. The threads within this SM traverse the graph from the root, calculating local changes to the BC scores. Finally, each SM adds its changes to the global BC scores atomically. Since CTAs are executing independently they will not finish calculating their local scores simultaneously. Hence, the contention of resources for the atomic updates to the global BC scores is low.

One of the most significant factors in how fast the algorithm executes is the choice of graph traversal method. For a graph traversal at the level of a CTA for betweenness centrality, it has been shown that assigning threads to each edge rather than each vertex of the graph achieves greater memory throughput on the GPU [29]. Alternatively, the use of an explicit queue can obtain even better performance for especially sparse graphs, such as road

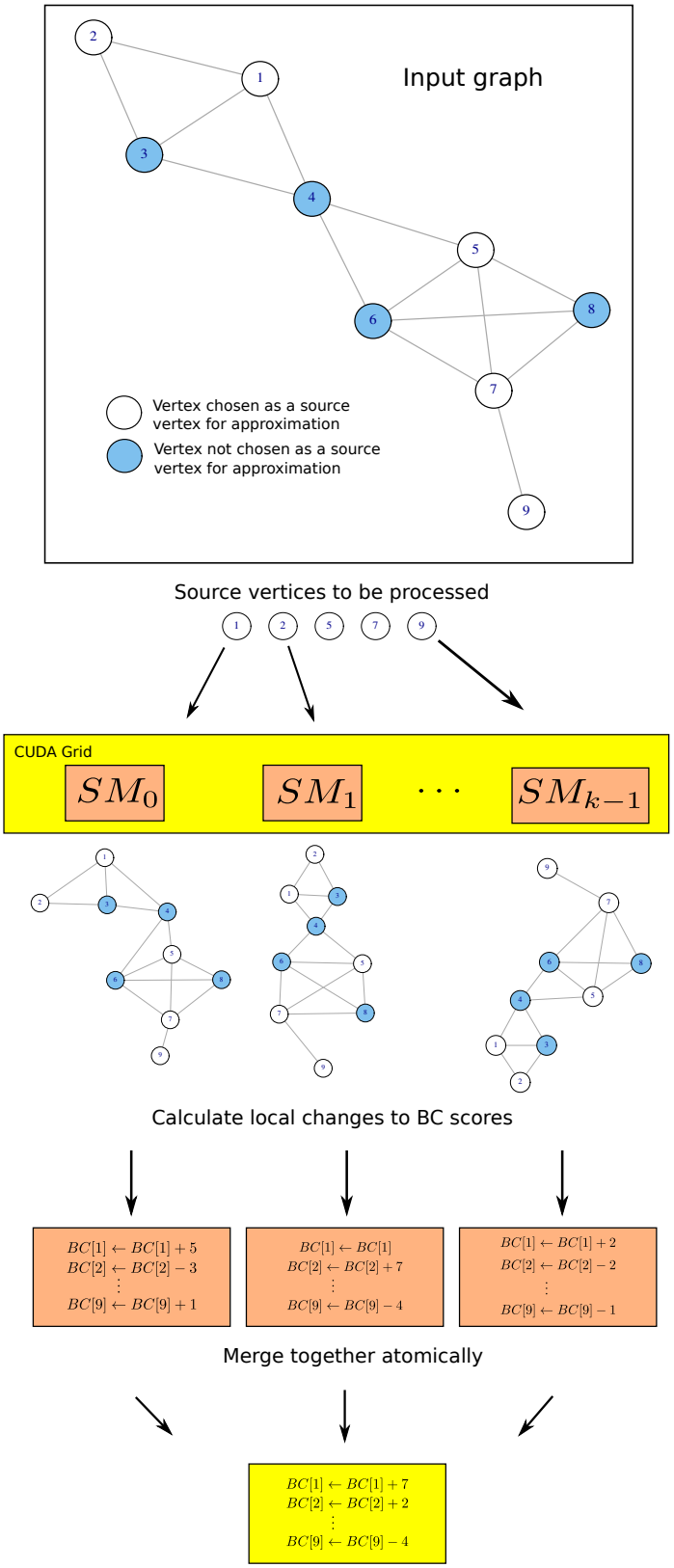


Figure 12: Decomposition of work to parallel compute units

Table 8: GPUs used for this study

GPU	*Tesla C2075*	*Tesla K40c*	*GT 640*
SMs	14	15	2
Memory (GB)	6	12	1
Frequency (GHz)	1.15	0.745	0.95
Compute Capability	2.0	3.5	3.5
TDP (W)	225	245	75

networks, or for dynamic updates to the graph that touch only a local subset of vertices [2]. Section 5.4 explores how these methods of parallelism impact the power consumption of the GPU.

5.3 Experimental Setup

Table 8 shows the various GPUs used for these experiments. The Tesla C2075 GPU is based on NVIDIA's "Fermi" architecture and cannot leverage the latest features of the CUDA programming model, such as Dynamic Parallelism; however, our implementations do not rely on such features. The Tesla K40c is NVIDIA's latest GPU designed specifically for HPC applications and is based on NVIDIA's "Kepler" architecture. These GPUs were designed with scientific computing in mind and have more memory than typical desktop GPUs. The GT 640 is a commodity GPU that is a part of the NVIDIA Kayla platform, an embedded system consisting of an NVIDIA Tegra 3 ARM Cortex A9 Quad-Core processor and the GT 640 GPU.

Algorithms were implemented in CUDA C++ using the CUDA 5.5 toolkit. Static computations were implemented to compute exact centrality scores whereas dynamic computations were implemented to also compute approximations to centrality scores using $k = 256$ randomly chosen roots as suggested by the DARPA SSCA benchmark suite [65]. We simulate dynamic graphs by randomly choosing 100 edges, removing them from the graph, and reinserting them sequentially, updating the BC scores after each insertion. This is the limit for low-latency applications that must respond to changes rapidly.

On the Kayla platform, power was measured using a Watts Up wall-plug meter, which

Table 9: Graph datasets used for this study

Graph	Nodes	Edges
as-22july06	22,963	48,436
astro-ph	16,706	121,251
caidaRouterLevel	192,244	609,066
coPapersCiteseer	434,102	16,036,720
delaunay_n12	4,096	12,264
delaunay_n14	16,384	49,122
delaunay_n20	1,048,576	3,145,686
eu-2005	862,664	16,138,468
kron_g500-logn16	55,321	2,456,071
kron_g500-logn19	524,288	21,780,787
luxembourg.osm	114,599	119,666
preferentialAttachment	100,000	499,985
smallworld	100,000	499,998

measures system power. Since the entire computation is executed on the GPU, the CPU is idle and its power is constant and small enough to be neglected. Power was sampled at one second intervals and averaged over the lifespan of a kernel. Typical edge updates take a small number of seconds and the instantaneous power does not significantly change throughout a kernel execution. Power on the Tesla GPUs is measured directly using the NVIDIA Management Library (NVML). This library provides a C-based API for measuring power and temperature of Tesla GPUs. We sampled power at 10 ms intervals and report the average of the lifespan of a kernel.

Finally, Table 9 shows the graph datasets used for this study. These graphs were obtained from the DIMACS Challenge archives [47] and represent a diverse set of networks ranging from planar road maps (*luxembourg.osm*) to power-law graphs representing the structure of web domains (*eu-2005*).

5.4 Experimental Results

5.4.1 Static Experiments

Since graph algorithms are memory bound, the faster that they can traverse edges the faster they tend to execute. Analgous to FLOPS for compute bound applications is the notion of Traversed Edges per Second, or TEPS. For an instance of betweenness centrality, the

Table 10: Energy-efficiency of static BC computations on the GPU for various classes of networks

Graph	Avg Power (W)	MTEPS/W
delaunay_n20	129.38	0.85
luxembourg.osm	95.41	0.35
preferentialAttachment	127.18	1.33
smallworld	127.10	2.54

number of TEPS is defined as follows [45]:

$$TEPS_{BC}(G,t) = \frac{mn}{t} \tag{10}$$

where n is the number of graph vertices, m is the number of graph edges, and t is the time in seconds. Defining a single work amount, here mn, regardless of the implementation is equivalent to defining the FLOPS for LU factorization as $2n^3/3$ regardless of the matrix arithmetic operations [78].

For the approximation of BC, n is replaced with k in defining TEPS. Table 10 shows the average power consumption and million of TEPS per W (MTEPS/W) for four different classes of graphs: meshes (*delaunay_n20*), road networks (*luxembourg.osm*), scale-free networks (*preferentialAttachment*), and networks with a diameter that is logarithmic in the number of vertices (*smallworld*). The TEPS/W metric is used to rank the energy-efficiency of graph processing systems for the Green Graph 500 [79]. These results were recorded using NVML and a Tesla K40c GPU. We can see that the *luxembourg.osm* road network consumed significantly less power on average than the other classes of graphs. Road networks tend to be extremely sparse and have very consistent degree distributions. In fact, no vertex (i.e. intersection) in this particular road network has more than 6 neighbors (i.e. incoming roads). As a result, each iteration of a breadth-first search over this graph results in a small amount of new vertices to be explored and consequently, few warps of execution per CTA and lower power consumption. Note that for all classes of graphs there isn't enough computation for the average power consumed to be anywhere near the TDP of the device.

Table 11: Comparison of dynamic BC computations on the CPU and GPU of the Kayla platform

Graph	*delaunay_n12*	*kron_g500-logn16*
Solution Quality	Exact	Approx. (k = 256)
CPU Time (s)	35.44	33.79
GPU Time (s)	1.32	1.33
Speedup	***26.92***×	***25.39***×
Average CPU Energy (J)	914.35	875.08
Average GPU Energy (J)	42.64	43.79
Energy Savings	***95.3%***	***95.0%***
CPU MTEPS/W	0.05	0.72
GPU MTEPS/W	1.18	14.37

Table 10 shows that our algorithm is more power-efficient on scale-free and small-world graphs. For these graphs we use an edge-based graph traversal to maximize the memory throughput of the GPU rather than using an asymptotically optimal traversal algorithm. These graphs tend to have a smaller number of traversal iterations that each contain tens of thousands of edges to traverse in parallel. In contrast, networks with larger diameters tend to have hundreds of edges to traverse per iteration, making it more challenging to fully utilize the GPU.

5.4.2 Dynamic Experiments

For dynamic calculations we compare against two baselines. First we can compare CPU and GPU implementations of the dynamic BC algorithm to see the benefit of using a massively parallel architecture. Second we can compare the dynamic GPU approach to a static GPU approach to see the benefit of updating analytics rather than recomputing them.

Table 11 compares using the CPU and GPU for computing BC scores dynamically. Note that the CPU algorithm is sequential. The times recorded represent the average time to update the BC scores for 100 edge insertions (one update occurs per edge insertion). Although the GPU requires slightly more instantaneous power than the CPU, we can see that the throughput provided by the GPU more than makes up for this additional power cost. The GPU implementation uses 19.69× less energy on average than the CPU for the

Table 12: Comparison of static and dynamic BC computations on the GPU of the Kayla platform

Graph	*delaunay_n12*	*kron_g500-logn16*
Solution Quality	Exact	Approx. (k = 256)
Static Time (s)	12.63	5.63
Dynamic Time (s)	1.32	1.33
Speedup	***9.6***×	***4.2***×
Static Energy (J)	424	188
Dynamic Energy (J)	42.6	43.8
Energy Savings	***90.0%***	***76.7%***
Static MTEPS/W	0.12	3.34
Dynamic MTEPS/W	1.18	14.37

two graphs above. Since these results were obtained on the Kayla platform, we had to restrict our analysis to significantly smaller data sets (and hence used approximation for the Kronecker *kron_g500-logn16* graph).

Using the same graphs, we compare static and dynamic methods for betweenness centrality in Table 12. The static implementation used as a reference here is from Jia *et al.* [29] (previous state-of-the-art) and the dynamic implementation is our own. Note that this static implementation differs from the one used in Table 10, which was our own implementation that improves upon the results from [29]. The times presented are again averaged over all 100 edge insertions. Although the time required for each update is highly dependent on the amount of work required by that update, even the slowest updates are faster than static recomputation. In addition to being faster than the static approach, the dynamic approach also consumes less power. The intuition behind this result is that a static computation of BC scores for the updated graph is an upper bound for the amount of work required by a dynamic update. Since the dynamic update only traverses edges that are affected by the update it avoids unwarranted accesses to memory. Overall, our dynamic method sees a 6.9× average speedup compared to a static recomputation for these two graphs. Dynamic updating consumes an average 83% less energy than static recomputation.

The insertion of an edge into the graph presents one of three possible scenarios from each root. The inserted edge can either connect vertices that are the same distance (Case 1),

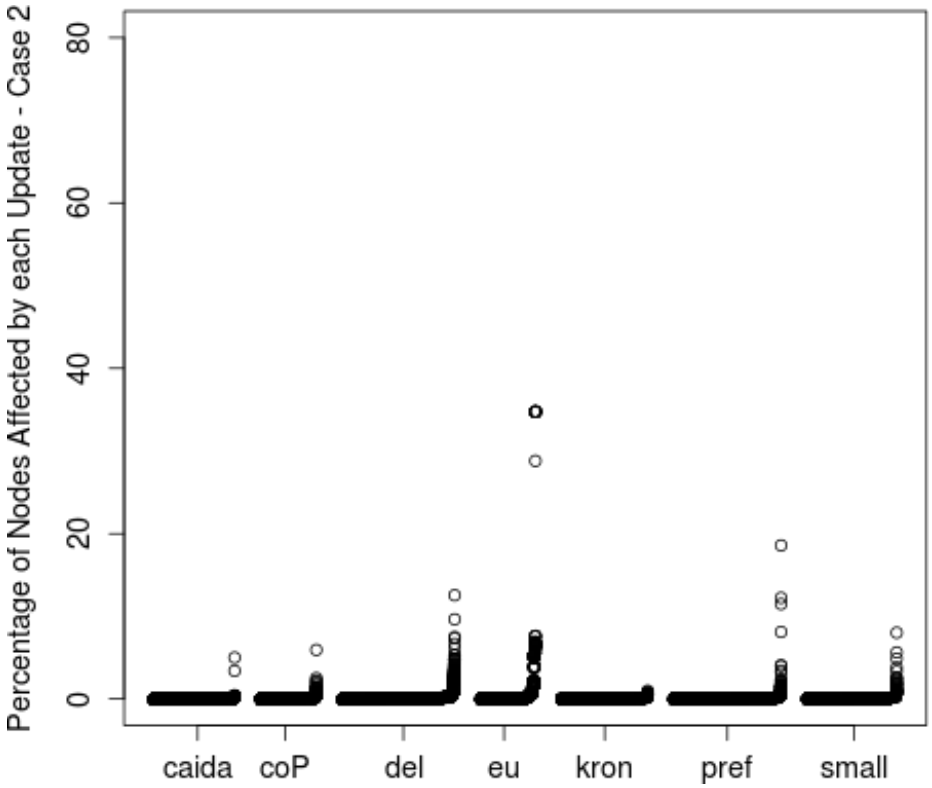


Figure 13: Percentage of vertices touched by Case 2 scenarios

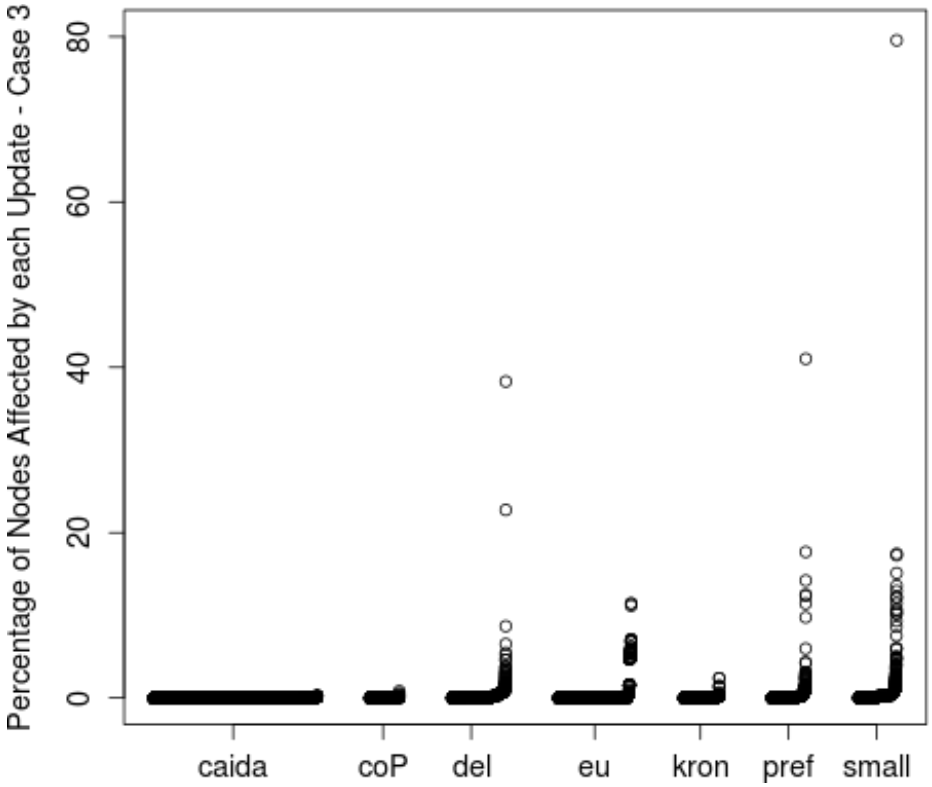


Figure 14: Percentage of vertices touched by Case 3 scenarios

adjacent distances (Case 2), or non-adjacent distances (Case 3) from a given root. Case 1 insertion scenarios do not change BC scores whereas Case 2 and Case 3 insertion scenarios require additional computation to account for the newly inserted edge [60]. To quantify how much work is required by the dynamic algorithm for a typical edge insertion, we record the percentage of vertices that are touched by the shortest path recalculations and dependency accumulations for each edge insertion. Figures 13 and 14 sort and display these percentages as a scatterplot for Case 2 and Case 3 insertion scenarios, respectively. Amazingly, a vast majority of edge insertions impact less than 1% of vertices in the graph. Out of the 62,844 Case 2 scenarios encountered, no more than approximately 35% of vertices were touched by any of them. Similarly, for Case 3, which tends to have more work as it pulls up vertices from further away from the root than Case 2 does, only three scenarios touched more than 30% of vertices in their respective graphs. This result implies that the use of asymptotically efficient algorithms is crucial to obtaining high performance for dynamic graph analytics.

To illustrate the effect of using a dynamic approach in terms of power, Figure 15 shows a scatter plot of the average power consumption during each edge update for two methods of parallelism for three graphs. The edge-based parallel method was introduced by Jia et al. [29] and assigns a thread to each edge of the graph to be inspected during each iteration of the graph traversal. The node-based parallel method instead uses an explicit queue to only traverse edges coming from vertices that are at the current depth of the graph traversal. The solid lines in the figure represent the average power consumption across all edge insertions. Since the edge-based parallel method checks every edge at every iteration of the search, it causes unecessary branching overhead and fetches to global memory. Since the edge approach does this unnecessary work regardless of where the edge is inserted into the graph the variance in power for the edge-based approach is small as the GPU consistently draws significant power. While this may normally be a sign that the processor is utilized in this case the processor is being fed superfluous instructions.

In contrast, the average power consumption for the node-based parallel method varies

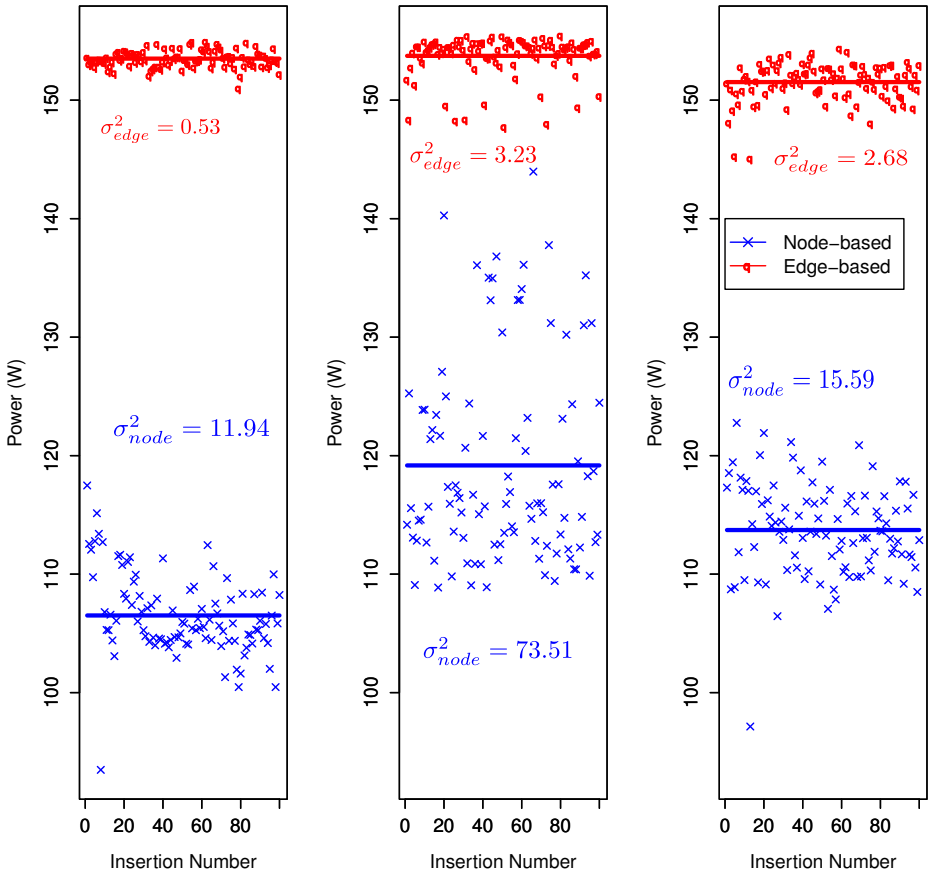


Figure 15: Left: *preferentialAttachment* Middle: *kron_g500-logn19* Right: *smallworld*

greatly with insertion. Intuitively, each edge insertion has some variable cost in terms of the portion of the graph that is affected by each update. Since the work done by the node-based method depends entirely on this cost, the power consumed by the node-based method is also variable. Note that in all cases tested the edge-parallel method consumes more power than the node-parallel method.

Finally, the above results are consistent regardless of the graph tested. The left portion of Figure 15 shows results for a scale-free graph, the middle portion shows results for a Kronecker graph, and the right portion shows results for a small-world graph. In each case the power consumption for the node-based method is significantly smaller and more volatile than for of the edge-based method.

5.5 Conclusions and Future Work

This chapter presents performance and energy efficiency optimizations for static and dynamic betweenness centrality. Our static implementation of the algorithm is capable of exceeding 2 MTEPS/W. Our dynamic implementation of the algorithm achieves greater than a 25× speedup over existing sequential methods on the CPU and a 6.9× average speedup along with an 83% average reduction energy-to-solution compared to a static recomputation of the analytic on the GPU. Our methods have been shown to work well on both embedded systems such as the Kayla platform and HPC systems such as Tesla GPUs. Furthermore, our methods are easily scalable to multiple GPU nodes for even faster processing.

Both parallel optimization as well as dynamic updating prove important to reducing total energy consumption. Applying these techniques to other algorithms may drastically increase the range of applications for graph analysis. More in-depth models of concurrency and energy consumption can guide analytic development. With sufficient hardware flexibility and programming models to match, advanced analysis of dynamic graphs will move from machine rooms to embedded and hand-held devices.

CHAPTER 6

FAST EXECUTION OF SIMULTANEOUS BREADTH-FIRST SEARCHES ON SPARSE GRAPHS

Graph analysis is a useful abstraction that can be applied to a variety of problems including epidemiology [38], hardware verification [80], and the modeling of physical phenomena [25]. Real world graph analytics require both scalability to large graph instances and high-performance implementations. Plenty of individual graph algorithms have been successfully accelerated using GPUs [3, 10, 34]; however, these manual implementations tend to make code reuse difficult compared to their CPU counterparts. Code reuse is tremendously important, because its absence results in tremendous effort spent duplicating and extracting work that has already been completed. Attempts to solve this issue have presented a number of abstractions that fit certain classes of graph algorithms. The Gather-Apply-Scatter abstraction (GAS), for example, uses a generalized addition operator to pull in and accumulate information from neighboring vertices (gather), update active vertices (apply), and propagate updates to neighboring vertices (scatter) [81]. Traversal-based abstractions tend to hide the performance-sensitive details of graph traversal while requiring the user to provide application-specific functions for visiting vertices, edges, or sets of vertices or edges [82–84]. There has also been a significant effort to standardize classical graph algorithms in the context of linear algebra [85]. These abstractions tend to focus on the efficient execution of a single breadth-first search, so for problems requiring many such searches, these approaches miss out on opportunities for coarse-grained parallelism and thus, additional performance gains.

In this study we focus on graph algorithms requiring many breadth-first searches that can be executed independently in parallel. This focus isn't contrived; many analytics require searching from several (or even all) vertices in a graph for the purpose of path-counting or analyzing a graph from multiple perspectives. For example, querying which

sets of vertices can reach one another can be implemented through a series of breadth-first searches, one from each vertex in the set. The All-Pairs Shortest Path (APSP) problem finds the length of the shortest path between every pair of vertices, which has been used to trace routes in transportation networks [86]. Betweenness centrality, a popular analytic for determining the most influential vertices in a graph, builds upon the APSP problem and thus also fits into this paradigm. The diameter of a graph as well as other useful graph metadata can also be determined from the solution of these problems.

We present the *multi-search* abstraction, which is a simple methodology for formulating algorithms that execute many simultaneous breadth-first searches. By providing a small number of typically short functions, the user can define his or her own algorithms that leverage this abstraction and our efficient implementation. We consider the multi-search abstraction to be a complement rather than a replacement for GAS and traversal-based paradigms. Although existing paradigms can be used to implement algorithms fitting the multi-search abstraction, we show that taking advantage of coarse-grained parallelism offered by performing many BFSs at once leads to better performance.

Based on the above, this chapter presents the following contributions:

- We present the *multi-search* abstraction, a simple, yet efficient methodology for expressing algorithms that execute many graph traversals.
- We provide an efficient, cooperative implementation of this abstraction, and show that it outperforms existing implicit GPU methods by greater than a factor of 2 for large graphs of varying diameter.
- Using our abstraction, we show that a single GPU can be used to solve the APSP problem on sparse graphs with millions of vertices whereas prior art required large distributed systems to scale to graphs of similar size.
- We implement betweenness centrality using our abstraction and show more than a 5.82x average speedup over existing parallel CPU frameworks, a 3.07x average

speedup over an existing GPU framework, and a 2.24x average speedup over a manual, heavily optimized GPU implementation for a diverse set of graphs.

6.1 Background

6.1.1 Terminology

A graph $G = (V, E)$ is an abstract representation of data consisting of a set of vertices V and a set of edges E connecting pairs of vertices. Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges in the graph. The work in this chapter focuses on graphs with uniform edge weight, but can be extended to handle graphs with arbitrary edge weights. For simplicity, we consider graphs with undirected edges here, noting that our implementation can handle graphs with either directed or undirected edges. An undirected edge between vertices u and v can be represented as two directed edges, one from u to v and the other from v to u.

A *path* p from a vertex u to a vertex v is a set of edges starting at u and ending at v. The *degree* of a vertex u is the number of edges incident to u or the number of vertices neighboring u. The *diameter* of a graph is the length of the longest shortest path between any two vertices. The vertices of a *scale-free* graph exhibit a power-law degree distribution such that a small number of vertices have a large number of neighbors and a large number of vertices have a small number of neighbors [15]. Graphs exhibiting the *small world* phenomenon (also known as six degrees of separation) have a diameter that is logarithmic in the number of vertices [40]. Such graphs are often, but not necessarily, scale-free. Finally, a *vertex frontier* is a subset of vertices that are currently active during an iteration of a graph traversal and an *edge frontier* is the set of outgoing edges from the current vertex frontier.

6.1.2 The Multi-Search Abstraction

At a high-level, the multi-search abstraction fits any problem that requires multiple, independent breadth-first searches. We consider it a generalization of traversal-based approaches, which abstract the details of graph traversal from the operations that need to be

performed on vertex frontiers. This abstraction allows domain experts in parallel algorithm design to construct the performance sensitive code that handles graph traversals and allows end users to write a small number of functions that are applied to the active vertices at each level. The end user may still need to be aware of some details of parallel programming, such as when atomic operations are necessary to avoid race conditions, but their code is typically concise and not substantially performance sensitive.

The users of our abstraction declare the set of vertices in the graph that traversals are to be executed from in addition to defining the functions that would be used in a standard traversal-based abstraction. Hence, the multi-search abstraction leverages coarse-grained parallelism because each search can be executed independently as well as fine-grained parallelism because the searches themselves can also be parallelized. In the context of GPU computing, this method of abstraction is especially useful for graphs that are too sparse to fully occupy the GPU with a single breadth-first search.

6.1.3 Multi-Search Algorithms

This subsection describes several fundamental graph algorithms that can be built on top of the multi-search abstraction. Many of these algorithms are used as subroutines themselves, showing the variety of use cases for the abstraction.

6.1.3.1 All-Pairs Shortest Paths

The All-Pairs Shortest Paths (APSP) problem finds the shortest paths between all pairs of vertices in a graph. The results can be represented as either the specific distances between each pair of vertices or as the paths themselves. For the latter representation, each vertex can store its parent in the breadth-first search tree that starts from the source. Note however, that the latter representation is nondeterministic as multiple valid parents may exist. For our experiments the choice of representation has a negligible impact on performance, so we choose to compute distances as has been done in other work in this area [87–89].

A canonical approach to solving the APSP problem is to use the Floyd-Warshall (FW)

Algorithm 14: The Floyd-Warshall Algorithm

```
1 for k ← 0...n − 1 do
2     for i ← 0...n − 1 do
3         for j ← 0...n − 1 do
4             d[i][j] ← min(d[i][j], d[i][k] + d[k][j])
```

algorithm, shown in Algorithm 14. Algorithm 14 assumes that $d[u][v]$ is initialized to 0 when $u = v$ and the weight of the edge $u \rightarrow v$ otherwise (or ∞ if no such edge exists). Having an $O(n^3)$ complexity that is independent of the number of edges in the graph makes this approach well-suited to dense graphs. In this chapter we instead focus our attention on sparse graphs, as graphs found in real world applications tend to be sparse [46, 49].

Algorithm 15: Simplified Version of Johnson's Algorithm for Sparse Graphs

```
1 for s ∈ V do
2     Q.enqueue(s)
3     while ¬Q.empty() do
4         v ← Q.dequeue()
5         for w ∈ succ(v) do
6             if d[s][w] = ∞ then
7                 Q.enqueue(w)
8                 d[s][w] ← d[s][v] + 1
```

For sparse graphs, a number of alternative approaches exist, such as Johnson's algorithm [90] or techniques based on Sparse Matrix-Vector Multiplication (SpMV) [91]. Johnson's algorithm repeatedly runs Dijkstra's Single-Source Shortest Paths (SSSP) algorithm, using every vertex in the graph as a source. For the graphs of uniform weight that we consider in this chapter, this algorithm can be simplified to have $O(mn)$ complexity [27]. Algorithm 15 shows a simplified version of Johnson's algorithm for graphs with uniform weight. The algorithm initially assumes that, for each source vertex s, $d[s][t] = 0$ when $s = t$ and that $d[s][t] = \infty$ otherwise.

Since shortest path calculations are independent from one source vertex to another, implementations choose to use a *chunk*, or number of source vertices, to compute at the same

time. Although the selection of a chunk size may or may not impact performance, it can have a tremendous impact on memory consumption. At one extreme, Floyd-Warshall approaches use a chunk of size n, as they compute shortest paths from all source vertices simultaneously, requiring $O(n^2)$ storage. At the other extreme, SpMV or BFS approaches uses a chunk of size 1, as they sequentially compute the shortest paths from one source vertex at a time, reusing $O(n)$ storage for the distances currently being computed (though the complete output still requires $O(n^2)$ space, of course). Our implementation is between these two endpoints: we use a chunk size equivalent to the number of streaming multiprocessors on the GPU, which is typically less than 16 on contemporary hardware. We can thus trivially scale our implementation to distributed systems with multiple GPUs per node by increasing the chunk size to be the total number of streaming multiprocessors across all GPUs on all nodes, as shown in [1] for betweenness centrality.

6.1.3.2 Diameter Computation

Computing the diameter of a graph once one has performed an APSP computation is trivial. The diameter can be defined in terms of d as follows:

$$diameter = \max_{u,v}\{d[u][v]\} \tag{11}$$

Although disconnected graphs have an infinite diameter, it is more helpful in practice to use a related metric, such as the diameter of the largest connected component or the effective diameter of some subset of the graph [92]. Knowledge of a graph's diameter is useful for designing the topology of computer networks in order to minimize latency, cost, and energy of sending messages between nodes [93]. Graph diameter has also been shown to have a significant performance impact for certain classes of parallel algorithms [1].

6.1.3.3 Transitive Closure

The *transitive closure* of a graph is the set of all reachable vertices from each vertex in the graph. Obtaining the transitive closure from d is also quite simple: if $d[u][v] \neq \infty$ then v is reachable from u, otherwise it is not. A number of trade-offs exist for computing

the transitive closure of a graph. Depending on the application and particular graph being analyzed, one may prefer to store the entire transitive closure as a matrix, using $O(n^2)$ space, but providing reachability queries in $O(1)$ time. For large graphs, one may instead prefer to perform a Breadth-First Search (BFS) from the source of the query in an attempt to find the destination of the query. This approach requires just $O(1)$ space for the result but takes $O(m + n)$ time for each query. Recent research has even proposed alternative methods that fall in between these two extremes [94]. Determining whether vertices can reach one another is a fundamental graph property that has been used for memory consistency verification [80], social network analysis [36], and the LU factorization of sparse matrices [95].

6.1.3.4 Betweenness Centrality

An example of an algorithm that requires more work per vertex yet still fits well into the multi-search abstraction is Betweenness Centrality (BC). Betweenness Centrality is a metric that attempts to determine the most influential or important vertices (or edges) in a network. The metric quantitatively measures importance by comparing the number of shortest paths passing through a particular vertex to the total number of shortest paths found in the graph. If we let $\sigma[s][t]$ be the number of shortest paths from a vertex s to another vertex t and $\sigma[s][t](v)$ be the number of those paths that pass through a third vertex v, we can define the BC score of v as follows:

$$BC[v] = \sum_{s \neq t \neq v} \frac{\sigma[s][t](v)}{\sigma[s][t]} \tag{12}$$

Brandes defined the *dependency*, which relates the BC scores of a vertex to its successors (from the perspective of a given source vertex s) [27]:

$$\delta[s][v] = \sum_{w \in succ(v)} \frac{\sigma[s][v]}{\sigma[s][w]}(1 + \delta[s][w]) \tag{13}$$

This recursive relationship allows for the computation of betweenness centrality in two steps of the multi-search abstraction. The first is a downward traversal from s that solves the

APSP problem and counts the number of shortest paths between all pairs of vertices. The second uses this information to sum dependencies between each pair of vertices back up the BFS tree until s is reached. The BC scores can be redefined in terms of the dependencies as follows:

$$BC[v] = \sum_{s \neq v} \delta[s][v] \tag{14}$$

With applications in electronic design automation, urban planning, and the analysis of the human brain, betweenness centrality has received much attention in recent literature [24, 25, 37]. BC has also been used a building block for more complicated algorithms, such as community detection [23].

6.2 Related Work

6.2.1 All-Pairs Shortest Paths

Noting that the APSP problem is a precursor to many other graph algorithms, it is not surprising that it has received significant attention in the literature. Prior implementations of the APSP problem tend to focus on dense graphs, distributed memory systems, and graphs of a relatively small scale (graphs containing fewer than 100,000 vertices). The APSP problem has been implemented on a rather diverse set of architectures, including FPGAs [96], GPUs [97], heterogeneous systems [89], and supercomputers [98]. We focus our work on GPU implementations of the APSP problem for large, sparse graphs representative of unstructured data found in real world applications [49].

Bondhugula *et al.* present an FPGA-based APSP implementation based on motivation from an application in bioinformatics [96]. They developed a tiled approach to solving the Floyd-Warshall algorithm, so their methods are most effective when applied to dense graphs. Katz and Kider implement the APSP algorithm on the NVIDIA G80 architecture, also using a tiled version of FW [97]. They present results for graphs with up to $n = 11,264$ vertices and show a method for handling graphs that are larger than the amount of memory provided by a single GPU. Buluç *et al.* use a blocked recursive elimination strategy to

solve the APSP problem on the GPU by noting that APSP corresponds to finding the matrix closure of the graph's adjacency matrix on the tropical semiring [87]. Using an NVIDIA GeForce 880 Ultra GPU, the authors provide an implementation that runs more than two orders of magnitude faster than an Opteron CPU. Matsumoto *et al.* provide yet another approach to computing the FW algorithm in a blocked fashion, this time on a heterogeneous CPU-GPU system [89]. Their results scale to graphs as large as $n = 43,776$ vertices, exceeding 1 TFLOPS in single-precision performance. Solomonik *et al.* implement a communication-avoiding block-cyclic APSP algorithm on a Cray XE6 supercomputer [98]. Djidjev *et al.* partition the input graph, solving the APSP problem independently on each component and unifying the results in a post-processing stage [88]. Although they focus on planar graphs, the authors present results on graphs with millions of vertices on a cluster with hundreds of GPUs.

The collection of work above focuses on algorithms that require $O(n^2)$ intermediate storage space (i.e., they use a chunk of size n). This choice of chunk size causes scalability issues on shared memory systems, requiring the deployment of these algorithms on large distributed systems to analyze graphs similar in size to the ones we study in this chapter. Okuyama *et al.* instead use an approach similar to that of Johnson's algorithm and found that the cost of such an approach was that the edge distribution plays a fundamental role in the performance of the algorithm [99]. However, the presented results are shown for graphs with up to only $n = 32,768$ vertices. Our approach contributes to this area by scaling to much larger graphs with only a single GPU through the use of a smaller chunk size, achieving performance comparable to that of previous work on large distributed systems. For instance, Solomonik *et al.* solve the APSP problem for a graph with 65,536 vertices in roughly two minutes on a system with 1,024 nodes (24,576 cores) [98]. They neglect to mention the density or structure of this graph, but since their implementation is matrix-based, their performance should be roughly equivalent regardless of the graph's sparsity.

For a randomly generated Delaunay mesh and Kronecker graph of the same size, our implementation requires just 41 and 99 seconds, respectively, on a single GPU. Hence, we provide a cost-effective, scalable, and fast solution to the APSP problem for sparse graphs. Furthermore, since we design our implementation as a general abstraction, other problems can leverage its results.

6.2.2 Parallel Abstractions for Graph Analysis

Recently, a number of shared memory and distributed graph programming frameworks have been developed in order to abstract the complicated details of conducting high-performance graph algorithms on parallel architectures [82–85, 100]. Many of these frameworks were inspired in part by the Parallel Boost Graph Library [101]. These frameworks tend to employ abstractions in the context of linear algebra, graph traversal, or the Gather-Apply-Scatter (GAS) paradigm. Popularized by the GraphLab framework [81], the Gather-Apply-Scatter (GAS) abstraction executes algorithms by repeatedly applying three steps:

1. Gather: Collect information about vertices and edges that are adjacent to the active frontier.

2. Apply: Update the vertices in the active frontier based on the gathered information.

3. Scatter: Use these updates to determine the vertices and edges that belong to the next frontier.

Alternatively, traversal-based abstractions have the algorithm developer provide code that is applied vertices or edges in the current frontier and sets up the frontier for the following search iteration. Finally, linear algebraic abstractions formulate graph algorithms as operations on vectors and matrices. For instance, a breadth-first search can be represented as a SpMV between the adjacency matrix of the graph and a vector representing the active vertex frontier on the (min, +), or tropical, semiring. The GraphBLAS standardizes a common set of building blocks for graphs through the language of linear algebra [85]. Our abstraction differs from a sparse matrix product in that the user writes a function to visit vertices

instead of defining a semiring, and is perhaps more general because of this difference. We also consider it more intuitive for the user to write a function in terms of vertex frontiers than to define a semiring.

6.3 Multi-Search Implementation

Our work complements existing paradigms as it provides a related abstraction that focuses on problems that require many graph traversals that can be executed independently. Although this abstraction is stricter than the traversal-based method in that it can applied to smaller set of algorithms, we show its merit through its significant performance benefits. We view the multi-search abstraction as a generalization of the traversal-based abstraction: the more coarse-grained parallelism that is available, the more likely one will benefit through the use of the multi-search paradigm rather than the traversal-based paradigm. This section presents an efficient, cooperative implementation of the multi-search abstraction that can be used to develop a number of useful graph algorithms (such as the ones described in Section 6.1.3). Similar to GAS and traversal-based methods, the multi-search abstraction derives its utility from decoupling the complicated details of the underlying graph traversals from the specific updates that need to occur for the higher-level algorithm at hand. Hence, users are only required to implement a few small functions that can all utilize the same device kernel for graph traversals, encouraging code reuse and alleviating programmers from having to implement their own error-prone sets of parallel graph traversals.

Algorithm 16 shows our cooperative implementation of the multi-search abstraction. Careful implementation of the abstraction is rather important, since it will profoundly affect the performance of all the algorithms that leverage the abstraction. The functions `init()`, `visitVertex()`, `update()`, and `finalize()` are left to be implemented by the user for his or her specific use case. The `for` loop on Line 1 is executed in parallel across the Streaming Multiprocessors (SMs) of the GPU. In practice, the user's case may only require

Algorithm 16: Pseudocode for the Multi-Search Abstraction Kernel

```
   // Loop across SMs
1  for i ∈ S do in parallel
2      Q_curr[0] ← i
3      Q_curr_len ← 1
4      Q_next_len ← 0
5      init(i)
6      barrier()
7      while Q_curr_len ≠ 0 do
8          for v ∈ Q_curr do
               // Loop across threads
9              for w ∈ neighbors(v) do in parallel
10                 visitVertex(i, v, w)
11         swap(Q_curr, Q_next)
12         barrier()
13         Q_curr_len ← Q_next_len
14         Q_next_len ← 0
15         update()
16         barrier()
17     finalize(i)
```

a subset of vertices to search, so we define the variable S to be this user-defined set. For the APSP problem, $S = V$. The number of vertices in the graph vastly outnumbers the number of SMs on the GPU, so each SM sequentially processes many iterations of this for loop. The while loop on Line 7 is executed by all of the threads within each SM. We organize work at the warp level, where a warp is a group of (as of this writing) 32 threads that execute in lockstep within each SM. Initially we assigned each thread to its own queue element and had warps cooperatively process the adjacency lists of these elements one at a time. Although this approach sufficiently balanced the work among threads within each warp, it left an imbalance of work *between* warps. For sufficiently small queues one warp could be left processing the entire frontier while the other warps idle. Improving upon this approach, we implemented a dynamic scheduling policy that has warps asynchronously dequeue vertices in the current vertex frontier. Instead of being statically assigned explicit batches of vertices within each vertex frontier to process, warps dequeue the next vertex

after processing their current vertex. Hence, a warp only will idle when there is no work remaining in the current frontier.

The threads within each warp cooperatively process the edges outgoing from to the dequeued vertex collected by that warp. This cooperation leverages the `__shfl()` intrinsic introduced by NVIDIA's Kepler architecture. The shuffle intrinsic allows for fast communication within a warp without requiring the use of shared memory. For instance, `__shfl(x,y)` returns the value of `x` held by thread `y` to all of the other threads in the warp. Each thread in the warp traverses consecutive outgoing edges from that queue element.

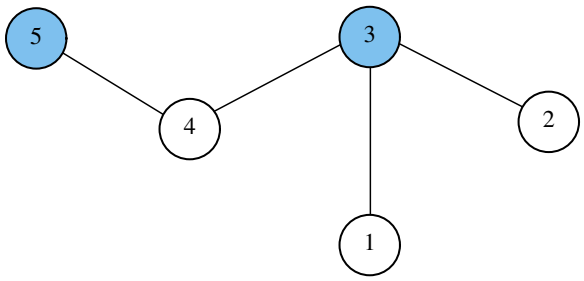
Algorithm 17: Implementation of `init()` for the All-Pairs Shortest Paths Problem

```
1 for k ∈ |V| do in parallel
2     if k = i then
3         d[i][k] ← 0
4     else
5         d[i][k] ← ∞
```

Algorithm 18: Implementation of `visitVertex()` for the All-Pairs Shortest Paths Problem

```
1 if d[i][w] = ∞ then
2     d[i][w] ← d[i][v] + 1
3     t ← atomicAdd(&Q_next_len, 1)
4     Q_next[t] ← w
```

Implementing the user-defined functions to implement the APSP algorithm on top of the multi-search abstraction is fairly straightforward. Algorithms 17 and 18 show APSP-specific implementations of `init()` and `visitVertex()`, respectively. The implementations for `update()` and `finalize()` can be left empty for this algorithm but are necessary for algorithms with additional stages, such as betweenness centrality. Algorithm 17 simply initializes $d[u][v]\ \forall\ u, v \in V \times V$. Algorithm 18 simply checks if w has been visited. If not, it is atomically added to Q_{next} to avoid race conditions. Note that duplicate entries in the queue are possible, since multiple threads may see w as unvisited before the first of



Work-efficient

$t_0 : (3,1), (3,2), (3,4)$

$t_1 : (5,4)$

Edge-parallel

$t_0 : (1,3)$ $t_2 : (3,1)$ $t_4 : (3,4)$ $t_6 : (4,5)$

$t_1 : (2,3)$ $t_3 : (3,2)$ $t_5 : (4,3)$ $t_7 : (5,4)$

Cooperative

$t_0 : (3,1)$ $t_1 : (3,2)$ $t_2 : (3,4)$

$t_{32} : (5,4)$

Figure 16: Several thread decompositions for the multi-search abstraction

these threads writes to $d[i][w]$. In practice, these duplicates are rare because they require either duplicate edges or multiple warps to simultaneously execute the same instruction. In our tests we found that the `atomicAdd()` on Line 3 was faster than having each warp prefix scan whether or not it found an unvisited vertex. This result makes sense because the atomic operation is with respect to a location in shared memory and because warps would have to perform their own scans since each warp expands a different queue element. For algorithms that require the *number* of shortest paths between each pair of vertices, an atomic Compare and Swap (CAS) operation must be used to set the distances of unvisited vertices. Otherwise, certain paths could be double counted, leading to incorrect results.

Several other methods for solving problems fitting this paradigm on the GPU are essentially algorithms that solve the APSP problem without the explicit use of this abstraction. For instance, Jia et al. present vertex and edge-parallel methods for computing betweenness centrality, noting that the edge-parallel approach better maximizes the memory bandwidth

achieved by the GPU [29]. Similarly, our prior work exhibits an approach that works particularly well for high-diameter graphs [1]. Both of these methods employ an approach to BC that reflects the multi-search abstraction in that the APSP problem is solved for each vertex before dependencies are computed. Hence, we can compare their mappings of simultaneous graph traversals to the GPU.

Figure 16 shows an example of how these methods differ for a simple graph. Consider a streaming multiprocessor that has been assigned a breadth-first search from vertex 4. In the first iteration of this search, vertex 4 will enqueue vertices 3 and 5, which become the active vertex frontier for the next iteration of the search. Figure 16 reflects this state, as vertices 3 and 5 are marked as active (shaded in blue). Beneath the picture of the graph in Figure 16 we show how the work-efficient approach [1], the edge-parallel approach [29], and our cooperative approach from Algorithm 16 assign the threads within the SM to edge traversals. The work-efficient approach assigns threads within each SM of the GPU to vertices on the active frontier. Hence, the first thread traverses the three outgoing edges from vertex 3 and a second thread traverses the outgoing edge from vertex 5. Note that although this method only traverses edges in the active edge frontier, threads have data-dependent amounts of work to do and hence the amount of work per thread can vary tremendously using this approach, leading to potentially severe load imbalances. The edge-parallel approach simply assigns a thread to every edge in the graph, regardless of whether or not it is in the active frontier. This approach easily occupies the GPU as many threads are needed to process iterations of moderate size; however, not all of these threads are contributing to the progress of the algorithm. Finally, our cooperative approach assigns warps to the adjacency lists of each vertex in the active frontier one by one. Hence, the three outgoing edges of vertex 3 are processed by 3 threads of the one warp and the lone edge from vertex 5 is then processed by one thread from a second warp, all in parallel. When graphs are sufficiently large such that the average adjacency list of a vertex tends to be greater than the warp size of the architecture, the utilization of each warp is high and the

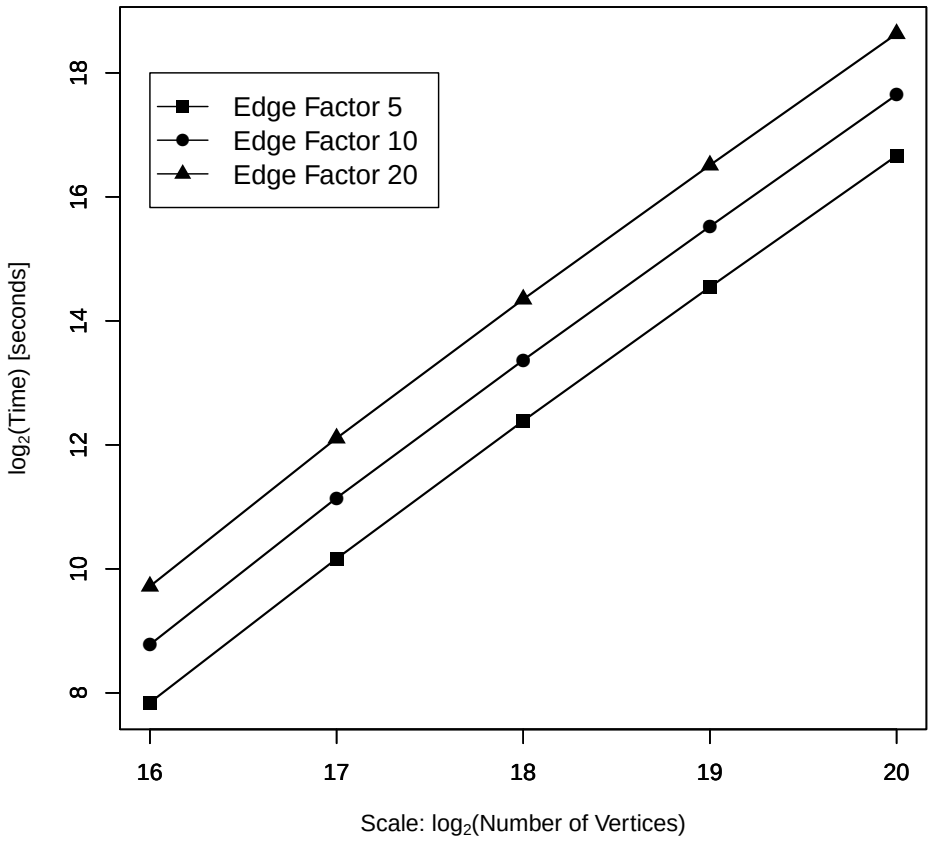


Figure 17: Impact of scaling vertices and edges on performance for Erdős-Rényi graphs

only edges processed by threads within each warp are edges in the current edge frontier. Furthermore, since this process is cooperative, threads have a well-partitioned amount of work.

Figure 17 shows the scalability of our cooperative approach presented in Algorithm 16 when it is used to solve the All-Pairs Shortest Paths problem. We use randomly generated Erdős-Rényi graphs and vary the number of vertices and edges to see how these changes impact performance. The edge factor influences the probability p of an edge selection in the $G(n, p)$ Erdős-Rényi random model [102]. Intuitively, a graph generated with twice the edge-factor will contain twice as many edges; however, the edge-factor should not be mistaken for average degree. The average degree of graphs with edge-factor 5 used to generate Figure 17 is approximately 28.

Figure 17 shows that our methodology is robust to scaling both the number of vertices

and the number of edges in the graph. On average, increasing the edge factor by two results in a 1.96x increase in execution time. One reason for why this increase in execution time is slightly less than the expected theoretical increase of 2x is that additional edges can provide better warp occupancy. For instance, if the degree of a vertex modulo the warp size of the architecture is small (but nonzero), additional edges will only give unoccupied threads work until all threads are occupied. Increasing the number of vertices by a factor of two (which, for the same edge factor, also increases the number of edges by a factor of two) results in an increase in execution time of 4.64x on average. Since the computational complexity of the algorithm is $O(mn)$, we theoretically expect to see a factor of 4 increase in execution time. The additional 0.64x increase that we see in practice could result from contention in resources when accessing memory atomically as well as from potential load imbalances among SMs.

6.4 Experimental Setup

In the next section we present performance results that show the scalability of the multi-search abstraction as well as how it performs for several classes of real-world graphs. To show the utility of the abstraction itself we implement betweenness centrality on top of it and compare the performance of our method to that of recent literature. CPU results were run on an Intel Core i7-2600K processor. The Core i7-2600K has a frequency of 3.4 GHz, and 8 MB last level cache, four physical processor cores and a peak memory bandwidth of 21 GB/s. We show results for CPU tests using 4 threads, since the use of hyperthreading didn't improve performance. GPU results were run on NVIDIA GeForce GTX Titan and NVIDIA Tesla K40c GPUs. The GeForce GTX Titan is a compute capability 3.5 GPU designed under the Kepler architecture that has 14 streaming multiprocessors, 6 GB of device memory, a clock frequency of 837 MHz, and a peak memory bandwidth of 288.4 GB/s. The Tesla K40c is another compute capability 3.5 Kepler GPU that has 15 streaming multiprocessors, 12 GB of device memory, a clock frequency of 725 MHz, and the same

Table 13: Graph datasets used for this study. Nodes and edges are displayed in millions.

Graph	Nodes	Edges	Notes	Sparsity
333SP	3.71m	22.22m	Ferrari	
adapative	6.82m	27.25m	Urban Sim.	
as-Skitter	1.70m	22.19m	Internet	
auto	0.45m	6.63m	Partitioning	
delaunay_n21	2.10m	12.58m	Triangulation	
ecology1	1.00m	4.00m	Gene Flow	
hollywood-2009	1.14m	115.03m	Movie Actors	
kron_g500-logn19	0.52m	43.56m	Kronecker	
ldoor	0.95m	45.57m	Large Door	
roadNet-CA	1.96m	5.53m	Intersections	
rgg_n_2_21_s0	2.10m	28.98m	Geometric	
thermal2	1.23m	7.35m	Diffusion	

peak memory bandwidth as the GeForce GTX Titan.

CPU code was compiled using `g++` version 4.8.1 and OpenMP. GPU code was compiled using `nvcc` and the CUDA 7.0 toolkit, which we leverage for C++11 support in device functions, allowing us to use lambda functions to implement the user-defined portions of the multi-search abstraction[1]. We present results based on publicly available graph data sets from the 10th DIMACS Challenge [47], the Stanford Network Analysis Platform [103], and the University of Florida Sparse Matrix Collection [46]. Table 13 shows more information about the set of graphs we perform tests on, including the number of vertices and number of (directed) edges for each graph, the significance of each data set, and finally the sparsity pattern of each data set. Note that we use both real-world and randomly generated graphs with highly varying connectivity from regular numerical meshes to irregular scale-free graphs.

For scaling experiments, we compare against the work-efficient [1] and edge-parallel

[1]Source code: https://github.com/Adam27X/graph-utils/

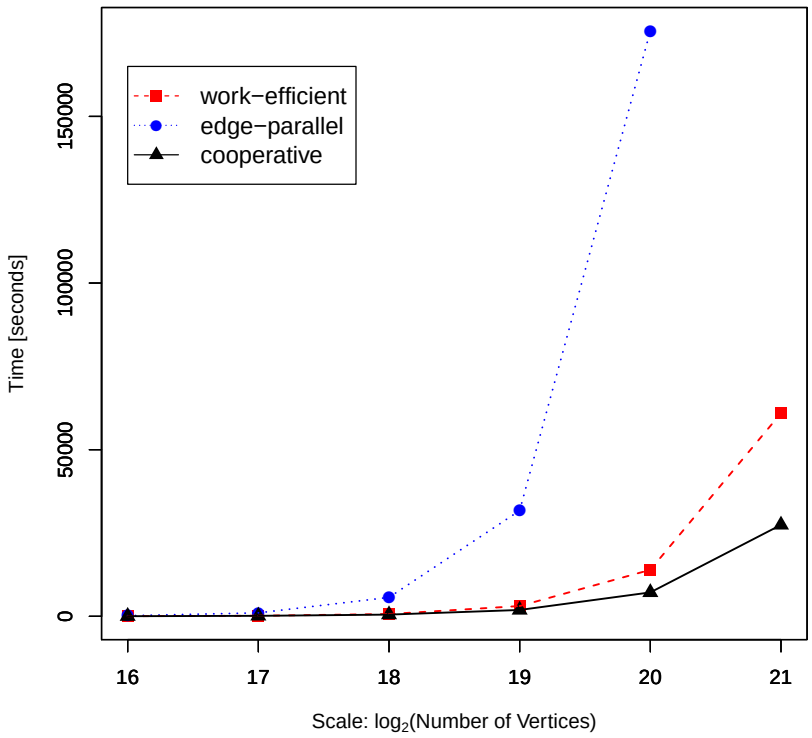
approaches [29] described in Section 6.3 and contrasted in Figure 16. These techniques are GPU-based and all of these experiments were run on the Tesla K40c GPU. For the experiments on the benchmarks in Table 13, we compare against both CPU and GPU implementations, where all GPU experiments were run on the GeForce GTX Titan GPU.

6.5 Experimental Results

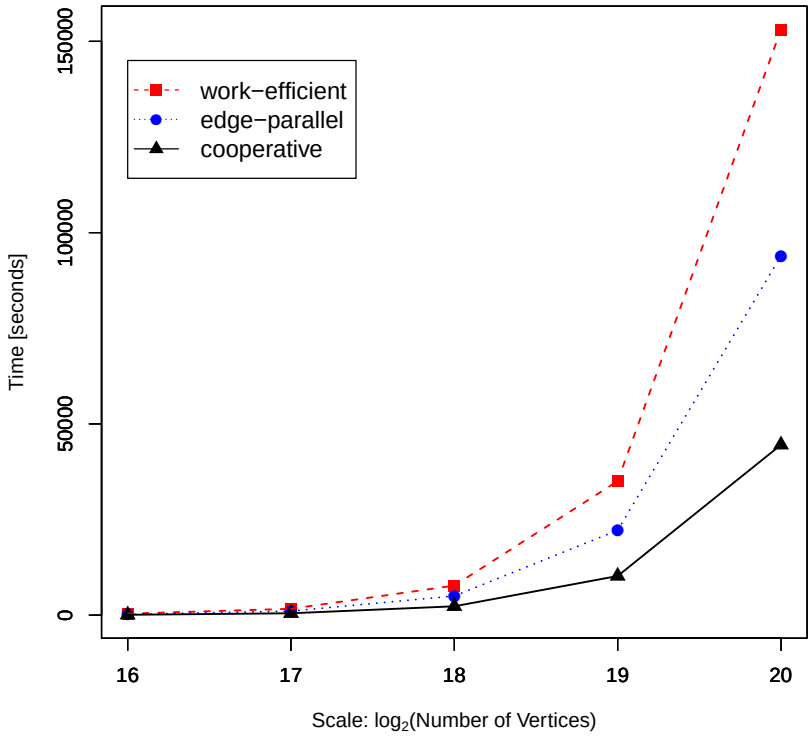
6.5.1 All-Pairs Shortest Paths

Figure 18 compares APSP execution times using the three methods of graph traversal shown in Figure 16. Figure 18a shows results for a high-diameter Delaunay mesh, which typically requires hundreds of search iterations to find all reachable vertices from a given source vertex. In contrast, Figure 18b shows results for a low-diameter, scale-free Kronecker graph, which typically requires fewer than ten search iterations to complete a Breadth-First Search. We can see that for the Delaunay mesh, the work-efficient approach from [1] is preferential to the edge-parallel approach from [29] whereas for the Kronecker graph, the opposite is true. This notion that the graph structure has significant performance implications lead to the hybrid approach presented in [1]. However, we can see that for both of these classes of graphs, our cooperative approach from Algorithm 16 is more robust to the structure of the graph, performing more than twice as fast on large scales of both high and low-diameter graphs. The work-efficient approach performs poorly on scale-free networks since the power-law distribution of vertex degree leads to severe load imbalances among threads. The edge-parallel approach, in contrast, does better on scale-free networks because a larger percentage of edges are active at once and since threads have an equivalent amount of work. However, this approach performs poorly on the smaller vertex frontiers seen in high-diameter graphs since a majority of threads will be assigned to edges that don't actually need to be inspected. The cooperative approach alleviates these issues by having the warps within each SM asynchronously process adjacency lists, allowing for work-efficiency as well as sufficient load-balancing among concurrent threads.

Table 14 shows the time required to solve the APSP problem as well as the maximum



(a) Triangular mesh



(b) Scale-free network

Figure 18: Comparison of multi-search traversal techniques for two classes of networks

Table 14: Benchmark results for solving the APSP Problem.

Graph	APSP Time (s)	Maximum Degree
333SP	93150	28
adapative	342253	4
as-Skitter	28333	35455
auto	2291	37
delaunay_n21	27432	23
ecology1	9187	4
hollywood-2009	43082	11469
kron_g500-logn19	10236	80674
ldoor	7802	76
roadNet-CA	34358	12
rgg_n_2_21_s0	49991	37
thermal2	13349	10

degree for our set of benchmark graphs. We include the maximum degree simply to enhance the information given in Table 13, where we could not fit it. Using a single GPU, we can scale to significantly larger graph instances in comparison to existing methods, since our approach uses a relatively small chunk of simultaneous source vertices. Existing implementations tend to either use large distributed systems to scale to graphs this large [88, 98] or restrict their studies to smaller instances of graphs on shared memory machines [87, 89, 96, 97].

It is intriguing to note that our performance on *ldoor* is better than that of *kron_g500-logn19* despite the fact that *ldoor* is a slightly larger graph. The irregularity of Kronecker graphs makes them particularly challenging to process. Significant warp divergences may occur when one warp processes the vertex with the largest number of edges, causing other warps in the block to idle before moving on to the next vertex frontier. Splitting vertices with especially large frontiers into virtual vertices is one potential improvement that could alleviate this issue [104] that we intend to explore for future work.

6.5.2 Betweenness Centrality

Since we present this work as an abstraction that can be applied to a number of problems requiring many simultaneous breadth-first searches, it is important to show the performance

of such problems under our abstraction. We compare our approach to four implementations of Betweenness Centrality from recent literature: Ligra [82], Galois [84], Gunrock [83], and a recent hybrid GPU implementation from [1]. Ligra is a shared-memory CPU graph framework that uses a traversal-based abstraction that allows users to write graph algorithms that map over frontiers of edges and vertices (or subsets thereof). Galois is a CPU-based system that provides the user with parallel set iterators, allowing the user to write sequential code that specifies loops that should be run in parallel. Rather than using the bulk synchronous parallel model of execution, Galois uses worklists to implement asynchronous execution. Gunrock has a similar programming interface to Ligra, but is written in CUDA for execution on GPU backends. It includes an advance stage that visits the current vertex frontier as well as a filter stage that generates the next frontier. Ligra, Galois, and Gunrock provide their own implementations of betweenness centrality, which we use for our experiments. Finally, the hybrid_BC GPU implementation of betweenness centrality uses an on-line approach to determine whether a graph will benefit more from either the work-efficient or edge-parallel methods that were shown in Figure 16. In terms of programmability, Galois is the most general as it can implement any worklist-based algorithm, Ligra and Gunrock are specialized to traversal-based graph algorithms, and hybrid_BC is a manual implementation that is specialized to betweenness centrality alone. Our multi-search abstraction is meant for algorithms requiring many graph traversals, but could be specialized to act similarly to Gunrock or Ligra in the event that a sufficient number of traversals aren't available for the user's application.

Table 15 shows timing results for each of these baselines as well as our own cooperative approach. The last row shows the average speedup for our cooperative implementation over the other methods. For all tests we approximate BC scores using $k = 8192$ source vertices to make the running time of the algorithm more reasonable. The approximation simply performs APSP calculations and dependency accumulations from k source vertices rather than all of them, so the time to compute the exact BC scores is roughly $\frac{n}{k}$ times the time to

Table 15: Benchmark results for computing Betweenness Centrality. Times are in seconds. The fastest result for each graph is presented in bold.

Framework	*333SP*	*adaptive*	*as-Skitter*	*auto*
Galois	4651	7086	1167	637
Ligra	3005	3442	1241	665
Gunrock	1999	4851	N/A	161
hybrid_BC	781	993	518	407
Cooperative	**352**	**601**	**275**	**74**

Framework	*delaunay_n21*	*ecology1*	*hollywood-2009*	*kron_g500-logn19*
Galois	2004	906	2058	1868
Ligra	992	635	4318	623
Gunrock	712	1458	630	**406**
hybrid_BC	373	176	1591	522
Cooperative	**174**	**104**	**602**	523

Framework	*ldoor*	*roadNet-CA*	*rgg_n_2_21_s0*	*thermal2*
Galois	1240	1498	3518	1088
Ligra	1751	700	2808	899
Gunrock	395	N/A	N/A	277
hybrid_BC	621	403	1066	204
Cooperative	**183**	**145**	**399**	**115**

Table 16: Average speedup of the cooperative approach over existing frameworks.

	Galois	Ligra	Gunrock	hybrid_BC
Speedup of Coop.	**7.66x**	**5.82x**	**3.07x**	**2.24x**

compute the approximate scores.

Compared to the parallel CPU implementations of Galois and Ligra our implementation does very well, averaging 7.66x and 5.82x speedups, respectively. The results in comparison to Gunrock are more interesting in that they vary tremendously. Since Gunrock uses a chunk size of 1 (i.e. it only leverages fine-grained parallelism for BC), it does particularly poorly on graphs with low average degree, such as *ecology1* and *adaptive*. On the other hard, Gunrock performs well on graphs that do offer lots of fine-grained parallelism, such as *hollywood-2009*, where its performance is competitive with ours and *kron_g500-logn19* where its performance is even better than our own. The entries denoted "N/A" in Table 15 for Gunrock correspond to graphs that caused a memory access violation on the GPU. For the graphs that we could compare, our implementation was 3.07x faster on average than Gunrock. Finally, our GPU abstraction is competitive with GPU code that is specialized for computing BC scores. The hybrid_BC implementation is never significantly faster than that of our own yet for *ldoor* our approach is 3.41x faster. Overall, our cooperative approach is 2.24x faster than hybrid_BC on average and is much more easily leveraged for the development of algorithms that can take advantage of the multi-search abstraction. Even though hybrid_BC is specialized for BC, our approach is faster because of our efficient implementation of graph traversals shown in Algorithm 16 and Figure 16. Table 16 summarizes the results in Table 15 by showing the average speedup our cooperative approach attains over the methods that we compare to from prior literature.

6.6 Conclusions

In this chapter, we present and provide an efficient implementation of an abstraction for processing many simultaneous breadth-first searches in parallel on the GPU. We implement the

abstraction by enlisting the threads within each warp to cooperatively traverse the edges in elements in the active vertex frontier. This approach is more than twice as fast as previous GPU approaches that were used to schedule threads for simultaneous graph traversals for large graphs of both low and high diameter. Furthermore, our approach scales to graphs with millions of vertices using a single GPU whereas previous approaches used large clusters to solve problems of similar size in greater amounts of time. Finally, our abstraction can efficiently implement more complicated algorithms. We show that an implementation of betweenness centrality that leverages our abstraction achieves an average speedup of 7.66x and 5.82x over the Galois and Ligra multi-core graph frameworks, a 3.07x speedup over the Gunrock GPU graph framework, and an average speedup of 2.24x over a heavily optimized on-line GPU implementation of betweenness centrality.

The literature on parallel implementations of graph algorithms is beginning to shift from manual, hand-tuned implementations of specific algorithms to libraries that provide abstractions for certain classes of parallel algorithms. The appropriate choice of an abstraction depends on the problem that needs to be solved, the way in which each abstraction is mapped to hardware, and the graph being analyzed. We consider the definition of new abstractions and the unification existing abstractions into a general parallel graph analytics framework to be an exciting area of future work.

CHAPTER 7

AN ENERGY-EFFICIENT ABSTRACTION FOR SIMULTANEOUS BREADTH-FIRST SEARCHES

Graphs can represent diverse sets of data from social networks [105] to the structure of computer programs [106]. Problems in areas such as urban planning [37] and epidemiology [38] are well-expressed by graphs and solved by different graph traversal algorithms. The applications often use large data sets relative to the platform and can leverage massive parallelism and memory bandwidth in GPUs for efficient computation.

The difficulty of programming GPU kernels and the immature state of GPU software has made it difficult for end-users to leverage the contributions of the domain experts who spend months of time optimizing GPU applications. GPU kernels are typically written and manually optimized for peak performance on a particular architecture, input data set, or application. Furthermore, relatively little work has been done to study the energy consumption of GPU algorithms. A lack of Dennard scaling and the era of dark silicon [107] imply that knowledge regarding the energy consumption of algorithmic choices is more important than ever.

This chapter addresses issues in abstraction and efficiency with an abstraction for solving simultaneous graph traversals on the GPU that allows energy- and time-efficient, general implementations. The abstraction itself is simple. Users only write functions for parts of the graph traversal relevant to their target application. Error prone and performance-sensitive details of executing parallel graph traversals on the GPU is buried in the general implementation, allowing users to focus on application-specific functionality. We evaluate our implementation against a set of diverse graphs, ensuring that the performance of our abstraction is not specialized to certain classes of graphs. Finally, we show that performance efficiency translates to energy efficiency.

In summary, we present the following contributions:

- We present an abstraction for executing simultaneous graph traversals on the GPU that permits a hybrid implementation. Vertices with sufficiently high outdegree are cooperatively processed by an entire warp whereas vertices with fewer neighbors are handled by a single thread.
- On NVIDIA GPUs, our implementation maximizes warp utilization and uses dynamic scheduling of warps to tasks to load-balance warps. We show that the additional performance efficiency reduces energy requirements.
- From web-crawls to road networks, we show that our abstraction achieves better performance and energy-efficiency than existing methods in addition to being more general. Our implementation saves 42% time and 62% energy on average over an oracle that chooses the best existing method for the graphs we studied.

7.1 Background

The computational throughput and memory bandwidth of GPUs provides a significant advantage over conventional CPU architectures in terms of performance and energy-efficiency [3]. Unfortunately, major efforts spent in the development of GPU algorithms that achieve high processor utilization see little to no reuse. Programming abstractions not only allow for more modular code but also make it easier for users to reason about the problems they are trying to solve rather than the details of parallel algorithm design or hardware. The remainder of this section presents the *multi-search* abstraction, a simple abstraction we employ for the execution of simultaneous breadth-first searches on sparse data sets.

7.1.1 The Multi-Search Abstraction

A number of existing libraries provide abstractions in order to simplify the development of parallel graph algorithms without sacrificing performance [82–84, 105]. These libraries typically use traversal-based abstractions that handle the performance-sensitive steps of a breath-first search, such as gathering neighbors and appropriately partitioning work to threads. Users of these abstractions are only required to implement a small number of

functions that are specific to the problem that they are trying to solve. These functions typically handle what data structures need to be updated when vertices are visited as well as the initialization and termination of the algorithm.

Inspired by these techniques and findings from previous work of our own, we generalize these traversal-based abstractions in the event that many such breadth-first searches are required by the user. The *multi-search* abstraction fits any problem that can execute independent breadth-first searches. The semantics of each search are the same with the exception that the searches start from different sources and write their own output. Examples of classical graph algorithms that fit this abstraction well are the All-Pairs Shortest Path (APSP) Problem, Reachability Querying, Betweenness Centrality, and Diameter Computation. Although traversal-based approaches can be applied to all of these problems, existing frameworks neglect the available coarse-grained parallelism (found in the independent searches) that these problems provide and thus miss out on opportunities for performance improvements, particularly for high-diameter graphs.

Finally, there have been a number of studies on the power consumption and energy-efficiency of GPU applications. Our prior work presents a study on GPU optimizations for static and dynamic betweenness centrality that showed an 83% average reduction in energy-to-solution over prior techniques [3]. This approach scales well to clusters of GPUs, providing substantial energy savings on large distributed systems. Nagasaka *et al.* use a statistical approach to model the power consumption of GPU kernels using hardware performance counters [108]. Hong and Kim develop an integrated power and performance model for GPU kernels and show potential energy savings for memory-bound applications [109].

7.1.2 Related Work

Ligra [82], Galois [84], GraphLab [81], the Parallel Boost Graph Library (BGL) [101], and the Multi-Threaded Graph Library (MTGL) [110] are frameworks that all provide CPU-based abstractions for graph analysis. GraphLab takes a disk-based approach, improving

upon distributed frameworks such as Pregel [111]. Green-Marl takes a domain-specific language approach to graph processing by applying compiler optimizations that could not be applied to more general purpose programs [112]. Galois and Ligra are shared memory CPU approaches to processing large graphs in memory. Galois uses internal parallel data structures to asynchronously process worklists while Ligra uses a traversal-based abstraction that internally uses a hybrid method of graph traversal based on the density of the graph.

The GraphBLAS provides a set of primitives for graph processing in the context of linear algebra [85]. Users of the GraphBLAS define a semiring on which to perform sparse matrix products. For instance, the APSP problem can be solved on the tropical (min,+) semiring [113]. In contrast, users of our system instead define callback functions that are invoked when vertices are visited.

GPU efforts in the realm of graph analysis have mostly focused on manual, monolithic implementations of specific algorithms [1, 10, 87] although a number of frameworks have been proposed in recent literature [83, 114, 115]. Medusa was the first such approach, providing APIs that can act on edges and vertices [114]. The Gunrock library from Wang *et al.* improves upon this work with load-balancing techniques that significantly improve performance [83]. GasCL is an OpenCL graph framework that uses GraphLab's Gather-Apply-Scatter (GAS) abstraction [115]. Finally, the CUSP library focuses linear algebraic implementations of algorithms that operate on sparse data sets [116].

7.2 Methodology

Breadth-First Searches (BFSs) consist of a number of search iterations, beginning with the source vertex of the search. Each iteration explores the unvisited neighbors discovered by the previous iteration. We define a *vertex frontier* as the set of vertices to be explored during a specified iteration of a breadth-first search. Users of our abstraction define the set of vertices in which traversals are to be enacted from as a small number of functions:

- `init()`: Initialize data structures at the beginning of program execution.
- `prior()`: Handle any computation that may occur just prior to a search iteration.
- `visitVertex()`: When an edge (u, v) is traversed from source i, update the appropriate data structures in terms of u, v, and i.
- `post()`: Handle any computation that may occur at the end of a search iteration.
- `finalize()`: Handle any computation that may occur after all the searches have completed.

These functions are typically short and performance-insensitive. When writing these functions, users will have to be aware that synchronization will sometimes be necessary to avoid race conditions, a consequence that has been observed in existing frameworks [82, 83].

7.2.1 Implementation

Prior literature has presented a number of ways to solve the APSP problem (or other algorithms requiring its solution as a subroutine) on the GPU [1, 29]. A number of these implementations implicitly implement the multi-search abstraction to handle graph traversals for their particular use case. Our approach, in addition to being more general, improves upon the performance provided by these techniques through the use of warp-synchronous programming and a hierarchical queueing scheme. GPU computing involves distributing work to Cooperative Thread Arrays (CTAs) as well as to the threads within each CTA. Warp-synchronous programming leverages additional knowledge about how CTAs are mapped onto GPU hardware, namely the fact that each streaming multiprocessor of the GPU executes instructions in lockstep in groups of 32 threads (on current NVIDIA platforms) referred to as warps (using CUDA terminology). This execution model allows programmers to have warps cooperatively and asynchronously process sets of data from other warps, minimizing intrablock barriers and allowing dynamic scheduling of tasks. For instance, we assign each warp to a vertex in the active frontier of the BFS being processed by the

SM to which the warp belongs. Rather than statically assigning warps $\{0 \ldots k-1\}$ to elements $\{0, k, \ldots\} \ldots \{k-1, 2k-1, \ldots\}$ of the frontier we use a dynamic scheduling policy that has each warp grab the next unprocessed queue element (using atomic operations to prevent race conditions). Although atomic operations have been shown to have significant performance impacts [117], this particular usage of them has a negligible effect on performance: the memory location under contention resides in shared memory and a maximum of 32 threads (one thread per warp and a maximum of 32 warps per thread block) will ever try to increase the counter that points to the next queue element to be accessed. The use of dynamic scheduling is significant for scale-free graphs since idle warps can effectively steal work from the critical warp, providing better load-balancing between the warps of each SM. NVIDIA's Kepler (and newer) architectures provide the `__shfl()` intrinsic that allows for exchanging data between the threads of a warp without explicit synchronization. We also use the `__ldg()` intrinsic to leverage the GPU's read only data cache for certain loads from global memory. Of course, this is just one implementation for a set of hardware problems. Others are possible, and users of the abstraction won't need to change their code when implementations of the abstraction are improved.

7.2.2 Thresholding

Initial experiments with the above approach performed poorly on graphs containing many vertices of low outdegree. When an entire warp is assigned to a vertex with outdegree smaller than the architectural warp size, some threads within the warp will be idle. Hence, for vertices with sufficiently small outdegree, we assign a single thread per vertex to gather its neighbors. We use two distinct queues, one that consists of vertices with sufficiently small outdegree to be processed by a single thread (Q_{small}) and one that consists of vertices with a larger outdegree to be processed by an entire warp (Q). During each iteration of the search, the vertices in Q are processed by the warps in each SM followed by the vertices in Q_{small} by individual threads. In order to determine how small the outdegree of a vertex must be to be enqueued into Q_{small}, we use a threshold T. If the outdegree of a vertex is strictly

less than T it will be enqueued into Q_{small}, else it will be enqueued into Q. The Multi-Threaded Graph Library [110] uses a similar partitioning of vertices based on outdegree to avoid load imbalance on CPUs.

Algorithm 19: Pseudocode for the Warp-Thread Hybrid Multi-Search Abstraction Kernel

```
   // Loop across SMs
1  for i ∈ S do in parallel
2      if outdegree(i) < T then
3          Q_small_curr.enqueue(i)
4      else
5          Q_curr.enqueue(i)
6      init(i)
7      barrier()
8      while ¬Q_curr.empty() ∧ ¬Q_small_curr.empty() do
9          prior()
10         barrier()
11         for v ∈ Q_curr do
               // Loop across threads
12             for w ∈ neighbors(v) do in parallel
13                 visitVertex(i, v, w, Q_next, Q_small_next)
           // Loop across threads
14         for v ∈ Q_small_curr do in parallel
15             for w ∈ neighbors(v) do
16                 visitVertex(i, v, w, Q_next, Q_small_next)
17         move(Q_curr, Q_next)
18         move(Q_small_curr, Q_small_next)
19         barrier()
20         post()
21         barrier()
22     finalize(i)
```

Algorithm 19 shows a pseudocode implementation of our multi-search abstraction. On the GPU we use a pair of arrays, Q_{curr} and Q_{next}, to represent a single queue. Q_{curr} contains the vertices in the active vertex frontier whose neighbors will be explored during the current iteration of the traversal. Q_{next} contains the unexplored neighbors of vertices in Q_{curr} that

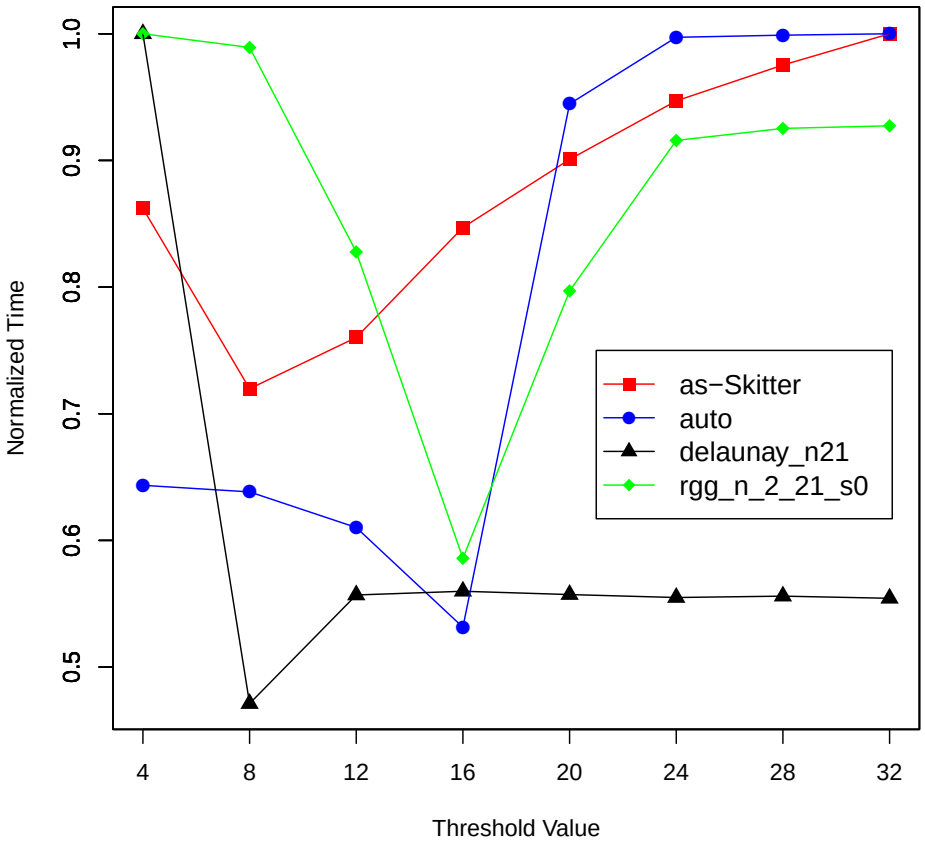


Figure 19: Relative effect of the threshold parameter T on performance.

will be explored during the next iteration of a traversal. We use two such queues to implement our hybrid approach: Q and Q_{small}. The `for` loops on Lines 11 and 12 process vertices with large adjacency lists sequentially by assigning an entire warp of threads to gather the neighbors of each active vertex. In contrast, the `for` loops on Lines 14 and 15 process vertices with small adjacency lists by assigning a single thread to sequentially gather the neighbors of each active vertex. The functions `init()`, `prior()`, `visitVertex()`, `post()`, and `finalize()` are user-defined functions to fit the higher-level application that they wish to target. These functions are typically concise and do not have significant impacts on performance. Depending on the user's application, some of these functions may even be left empty. For instance, to solve the APSP problem, only `init()` and `visitVertex()` need to be defined. Our implementation uses C++ templates to allow users to define these functions as functor objects, function pointers, or lambda expressions.

Table 17: Graph datasets used for this study.

Graph	Nodes	Edges	Significance
333SP	3,712,815	22,217,266	Ferrari
adapative	6,815,744	27,248,640	Urban Sim.
as-Skitter	1,696,415	22,190,596	Internet
auto	448,695	6,629,222	Partitioning
delaunay_n21	2,097,152	12,582,816	Triangulation
ecology1	1,000,000	3,996,000	Gene Flow
hollywood-2009	1,139,905	115,031,232	Movie Actors
kron_g500-logn19	524,288	43,561,574	Kronecker
ldoor	952,203	45,570,272	Large Door
rgg_n_2_21_s0	2,097,152	28,975,990	Geometric
roadNet-CA	1,971,281	5,533,214	Intersections
thermal2	1,227,087	7,352,268	Diffusion

Figure 19 shows the relative improvement in performance for varying values of T for a few graphs. Note that when $T = 0$, all vertices are placed into Q_{large} and are thus processed by an entire warp. At the other extreme, when $T = \infty$, all vertices are placed into Q_{small} and are thus processed by a single thread. From Figure 19 we can see that $T = 8$ and $T = 16$ lead to the largest improvements in performance, depending on the particular input. When the threshold is too low, vertices with low outdegree are processed by an entire warp, leading to many idle threads within the warp. Conversely, when the threshold is too high, vertices with high outdegree are processed by a single thread when they instead supply enough parallelism for an entire warp of threads, leading to load imbalances from critical threads that have to traverse large adjacency lists. We consider using a histogram of vertex outdegree for the entire graph as a method of dynamically determining an appropriate value of T to be an interesting idea for future work. For our complete set of graphs using the value $T = 16$ worked best overall, and this value will be used to report results in the next section.

7.3 Evaluation

7.3.1 Experimental Setup

Table 17 shows the input graphs used to evaluate our techniques. These graphs are publicly available data sets from the DIMACS Challenge archives [47], the University of Florida

Sparse Matrix Collection [46], and the Stanford Network Analysis Project (SNAP) [103]. We use both real world graphs, such as the *as-Skitter* Internet topology graph, and randomly generated graphs such as the *rgg_n_2_21_s0* geometric graph. These graphs have a broad range of diameters as well, which is important as graph diameter has previously been shown to significantly impact performance [3]. Graphs such as *delaunay_n21* have a high diameter, which leads to small vertex frontiers and many search iterations to completely traverse the graph. In contrast, graphs such as *hollywood-2008* have a low diameter, which leads to the majority of vertices being explored in a single search iteration and typically fewer than 10 search iterations to completely traverse the graph.

Code was written in CUDA C++ using the CUDA 7.0 toolkit. For timing experiments we use an NVIDIA GTX GeForce Titan GPU; since energy measurement via the NVIDIA Management Library (NVML) can only be used on Tesla GPUs, we use an NVIDIA Tesla K40c for experiments involving measurements of power and energy. Both the Titan and K40 GPUs are based on the "Kepler" architecture, have compute capability 3.5, a peak theoretical memory bandwidth of 288.4 GB/s, and a warp size of 32 threads. The Titan has 14 SMs, 6 GB of global memory, and a base clock frequency of 837 MHz. The K40 has 15 SMs, 12 GB of global memory, a base clock frequency of 745 MHz, and a TDP of 245 W.

Using NVML and C++11 futures, we spawn off a CPU thread to measure power asynchronously as GPU kernels of interest are launched[1]. We sample the power of the GPU once every ten milliseconds with a call to `nvmlDeviceGetPowerUsage()`, which will only work for Tesla class GPUs (hence our use of the K40). Using these samples we record the numerical integration of the sampled power consumption for the lifespan on a kernel, where one kernel launch is all that is necessary for all 1024 graph traversals for a given data set.

We evaluate our approach by comparing it to other GPU methods used to solve the APSP problem (or problems that it builds upon). We choose a value of $k = 1024$ source

[1] Source can be found at https://github.com/Adam27X/graph-utils/

vertices to perform graph traversals from in order to keep execution times reasonable. For graphs that compromise of one large connected component that contains greater than 90% of all vertices in the graph (such as many real world graphs [118]), the time to execute a graph traversal varies minimally from one source vertex to another. Hence, the conclusions we draw from our use of a subset of source vertices can be confidently applied to the computation of the entire APSP problem. The methods we compare to are as follows:

- `Edge-parallel`: Assigns a thread to every edge of the graph for every search iteration, regardless of whether or not the endpoints of that edge are vertices in the active frontier. This approach is most effective when many vertices belong to the active frontier [29].
- `Work-efficient`: Assigns a thread to every vertex in the active frontier [1]. When $T = \infty$, our hybrid approach simplifies to this approach.
- `Oracle`: A pseudo-hybrid approach that chooses between `Edge-parallel` and `Work-efficient`, depending on whichever method is better for the particular input graph being analyzed. We present this result as a proxy for the hybrid method used to compute Betweenness Centrality in [1].
- `Warp-based`: Our method that assigns a warp to every vertex in the active frontier. Threads within the warp process consecutive outgoing edges from the active vertex. When $T = 0$, our hybrid approach simplifies to this approach.
- `Warp-thread Hybrid`: Our method that uses two queues, one containing elements to be processed by a single thread and the other containing elements to be processed by an entire warp. When the outdegree of a vertex is less than T, we use a single thread to collect its neighbors. We use a static value of $T = 16$ for all experiments.

7.3.2 Experimental Results

Table 18 shows the time required to execute all 1024 graph traversals (in seconds) on the GeForce GTX Titan using the methods explained in the previous section. Although our `Warp-thread Hybrid` approach is best for 10 of the 12 graphs tested, the magnitude by

Table 18: Timings for various methods of graph traversal in seconds.

Graph	`Edge-parallel`	`Work-efficient`	`Oracle`	`Warp-based`	`Warp-thread Hybrid`	`Savings of Hybrid over Oracle`
333SP	1279.5	47.8	47.8	68.0	**32.1**	33%
adapative	7704.7	54.8	54.8	183.6	**42.8**	22%
as-Skitter	30.7	27.8	27.8	12.9	**9.66**	65%
auto	21.6	15.6	15.6	5.48	**4.82**	69%
delaunay_n21	436.8	23.3	23.3	25.1	**15.0**	36%
ecology1	426.9	8.86	8.86	29.1	**6.51**	27%
hollywood-2009	81.6	145.8	81.6	21.3	**20.4**	75%
kron_g500-logn19	35.4	55.1	35.4	**16.4**	17.1	52%
ldoor	434.0	**35.4**	**35.4**	35.7	36.5	-3%
rgg_n_2_21_s0	1824.9	67.0	67.0	37.0	**23.7**	65%
roadNet-CA	183.8	12.3	12.3	15.0	**9.15**	26%
thermal2	650.4	11.7	11.7	19.4	**7.71**	34%

which it is best is dependent on the threshold parameter T. For instance, our non-threshold based `Warp-based` approach does better than our hybrid approach for *kron_g500-logn19* and *ldoor*; however, simply setting $T = 0$ for such graphs would solve this issue. Hence, for future work we will consider an approach that determines the appropriate value of T for a given graph based on the distribution of the outdegree of its vertices.

Interestingly, for *ldoor* the `Work-efficient` approach is slightly better than both our `Warp-based` and `Warp-thread Hybrid` approaches. Again, setting the threshold dynamically (to $T = \infty$ in this case) could solve this problem. The maximum outdegree for any vertex of *ldoor* is 76 and 99.8% of vertices in *ldoor* have an outdegree of 63 or less. Since the current warp size of NVIDIA GPUs is 32 threads, this means any vertex assigned to a warp will process at most three (and very often only two) edges, which doesn't provide sufficient instruction level parallelism to each thread. Hence, assigning active vertices to threads for this graph results in marginally better performance. Overall, our `Warp-thread Hybrid` approach improves upon that of the `Oracle` by 42%. In practice, the implementation of such an oracle would have some overhead associated with choosing between the `Edge-parallel` and `Work-efficient` methods, making this result a lower bound for the

Table 19: Energy Consumption for various methods of graph traversal in Joules.

Graph	Edge-parallel	Work-efficient	Oracle	Warp-based	Warp-thread Hybrid	Savings of Hybrid over Oracle
333SP	36,676	2,220	2,220	2,880	**1,083**	51%
adapative	316,299	8,551	8,551	6,790	**1,433**	83%
as-Skitter	5,004	4,688	4,688	**1,053**	1,606	66%
auto	635	1,164	635	539	**159**	75%
delaunay_n21	12,403	1,425	1,425	1,495	**508**	64%
ecology1	12,309	856	856	1,642	**219**	74%
hollywood-2009	11,546	25,577	11,546	**1355**	3,342	71%
kron_g500-logn19	1,180	2,492	1,180	1,195	**570**	52%
ldoor	12,527	1,829	1,829	1,855	**1,210**	34%
rgg_n_2_21_s0	50,525	2,851	2,851	1,908	**759**	73%
roadNet-CA	26,950	2,050	2,050	**1,144**	1,539	25%
thermal2	17,566	996	996	1,311	**259**	74%

improvement of our approach in practice.

In addition to being faster than existing approaches, our `Warp-based` and `Warp-thread Hybrid` approaches tend to consume less instantaneous power. The `Edge-parallel` approach is energy-inefficient because threads are assigned to edges that don't necessarily belong to the active frontier and the `Work-efficient` approach is energy-inefficient because threads have an imbalanced amount of neighbors to gather and hence cause other threads within the same warp to stall and wait for whichever thread belongs to the warp's critical path. Table 19 shows the energy required to execute all 1024 graph traversals (in Joules) on the Telsa K40c using these approaches. For almost every graph we tested, the energy savings of our techniques are greater than the savings in time shown in Table 18, confirming the above analysis regarding the energy-efficiencies of prior work. Even though our performance results were slightly slower than the `Oracle` for *ldoor*, our energy usage is much better due to our efficient warp utilization. For *as-Skitter*, *hollywood-2009*, and *roadNet-CA*, we can see that there are interesting trade-offs between performance and energy consumption; although our hybrid approach provides the best performance for each of these graphs, our warp-based approach provides better energy-efficiency. The choice of

the threshold parameter T again plays a considerable role for these trade-offs. Overall, our `Warp-thread Hybrid` approach saves 62% energy on average compared to the `Oracle` approach and we again note that this figure neglects the energy cost of choosing a preferential distribution of threads to work that would be required by the implementation of such an oracle.

7.4 Conclusion

This chapter explored the performance and energy characteristics for multi-search, a simple GPU abstraction to execute simultaneous breath-first searches. Our initial approach of assigning warps to cooperatively gather neighbors from vertices in the active vertex frontier worked well for low-diameter graphs, but suffered from warp occupancy and utilization for high-diameter graphs. To account for this deficiency, we presented a hybrid approach that assigns a single thread to gather neighbors of vertices with sufficiently small outdegree. Across a varied set of real-world and synthetic graphs, our hybrid approach saves 42% time and 62% energy on average over an oracle that is an idealized representation of previous literature.

In addition to implementing a dynamic version of our hybrid approach we plan to consider performance and programmability tradeoffs to obtain a desirable level of abstraction for future work. The automation of GPU kernel optimization is another area of work that we consider to be important. Such automation can be achieved through compiler optimizations, runtime libraries, and even domain-specific languages.

CHAPTER 8

PARALLEL METHODS FOR VERIFYING THE CONSISTENCY OF WEAKLY-ORDERED ARCHITECTURES

Modern architectures use memory reordering techniques to obtain better performance and energy efficiency. For instance, high latencies to memory can be hidden by overlapping memory accesses with computation. Allowing for the reordering of memory instructions comes at the cost of design complexity, verification effort, and programmer burden [119]. On today's shared memory multiprocessor systems these problems are exacerbated by an increasing number of cores. Although techniques such as speculative execution, shared caches, coherence mechanisms, and instruction pipelining all have well-known benefits, an improper implementation of these techniques can lead to subtle memory errors such as data corruption or illegal instruction ordering [80]. Furthermore, the use of these techniques are visible to programmers, especially those concerned with the low-level, performance-sensitive details of the system [120].

Shared memory multiprocessor systems have a *memory consistency model* that is essentially a contract between hardware and software regarding the semantics of memory operations [121]. The simplest memory consistency model is the *Sequential Consistency* (SC) model. Under this model, all processors observe the same ordering of operations serviced by memory. Processors execute instructions precisely in the order specified by the program, or *program order*. A read from a particular location in memory is guaranteed to return the value of the last write to that location under the SC model. Although this model is intuitive, it restricts the use of performance optimizations commonly used by hardware and compiler designers [120].

In contrast, the ARM processors considered in this work have a significantly more relaxed memory model [122]. Weakly-ordered ARM processors allow speculative execution and reordering of a thread's reads and writes. Additionally, writes are not guaranteed to be

simultaneously visible to other cores. A consequence of these relaxations is that, the order in which instructions access memory (e.g., the *memory order*) on such processors is distinct from the program order. In comparison to the SC model, many more outcomes satisfying relaxed memory models exist, which makes direct verification a challenging process.

The verification process affects a processor's time to market: verification plays an important role in discovering defects early during the design process when remediation is less costly. As such, the problem of verifying that a multiprocessor complies with its memory consistency model has seen significant attention in the literature [80, 123–126]. Formal approaches attempt to exhaustively check a design using proof methodologies, but cannot scale to the size of current microprocessor designs that require millions of lines of RTL code [127, 128]. Furthermore, formal approaches tend to employ a high-level abstraction of a microarchitecture design, neglecting the details of its implementation. Unfortunately, the implementation itself is a significant source of bugs in large designs [128].

The verification of an execution against its system's memory consistency model is an NP-complete problem [121, 123]. Contemporary solutions thus trade time for accuracy, providing polynomial time approaches that are incomplete: they may miss violations of the memory model, but violations that are found are legitimate. We present our work in the context of TSOtool, a software package that employs a graph-based approach for verifying the Total Store Order (TSO) model [80]. TSOtool easily extends to other relaxed memory models, can evaluate specific processor implementations as well as generic protocols, and has been used to find subtle bugs in commercial products [80, 124].

Despite the usage of polynomial time verification algorithms, consistency verification is typically limited by both strong and weak scalability. Since these techniques are incomplete, test coverage is dictated by the number of program traces that can be evaluated. Practical scalability with respect to trace size is also important: it is desirable for instruction traces to comprise very long periods of race conditions and asynchronous behavior among parallel processors [126]. The bugs that existing tools are designed to find are deep

corner cases that slip through pre-silicon verification. Longer tests put caches and supporting logic in more interesting states that are likely to trigger such bugs, if they exist. A high-performance approach additionally allows verification engineers to tailor their tests to specific issues much more rapidly given results from prior tests. Hence it is desirable to execute larger tests as well as to execute a single test as fast as possible.

This paper addresses these challenges and presents the following contributions and results:

- We improve existing iterative, graph-based approaches for memory consistency verification by diminishing how frequently data structures need to be updated. This refinement reduces the work complexity of the algorithm from $O(n^2p^2d_{max})$ to $O(n^2p)$ per iteration for a program execution graph with n vertices, p virtual processors, and maximum vertex outdegree d_{max}. We prove that this reduction of work converges to the same result that would be computed by prior techniques.

- In addition to sequential speedups over existing approaches, our approach is more amenable to parallelization because it performs batched graph updates with less frequency. We implement parallel versions of our sequential approach in both OpenMP and CUDA and for sufficiently large test instances of interest, our GPU approach can achieve over an order of magnitude speed increase over our sequential approach.

- Although our optimizations are focused on a subset of the overall consistency verification problem, for large test cases our GPU approach achieves an average application speedup of 26.36x over a modified version of TSOtool used to verify ARM-based processors that we have been experimenting with at NVIDIA.

8.1 Background

The goal of our application is to verify the correctness of the memory subsystem as it is being designed, which implies that we need to ensure that the processor's memory consistency model is not violated. Based on dependencies between instructions of a program that are required to be satisfied by the rules of the architecture (such as read after write hazards), we can construct a partial ordering of memory instructions, which we model as a directed graph. Given the outcome of a specific execution of our program, we can *infer* additional edges that are required to be satisfied by the rules of the consistency model (such as ensuring that a load reads the most recently written data to memory). These inferred edges densify the graph representation, creating a more complete (but not necessarily total) ordering of memory instructions. If a cycle manifests from this process then we have a contradiction in the memory order and thus the memory model was violated or is invalid.

The remainder of this section provides more detail regarding the memory consistency verification process as implemented by TSOtool [80].

8.1.1 Constraint Graph

Let a graph $G = (V, E)$ consist of a set V of $n = |V|$ vertices and a set E of $m = |E|$ edges. A *directed edge* $(u, v) \in E$ originates from vertex u and terminates at vertex v. A *cycle* is a sequence $u_0, u_1, \ldots, u_k, u_0$ of vertices starting and ending at the same vertex (u_0) such that there exists an edge in G between each consecutive pair of vertices in the sequence. The *diameter* of a graph is the length of the longest shortest path between any pair of vertices.

Our method of consistency verification is concerned with *constraint graphs* [119, 129], partially-ordered directed graphs that model the memory semantics of a given program execution. The vertices of the graph represent dynamic processor instructions and the edges represent dependence relationships. Instructions have several key attributes:

- Instruction type (Load, Store, or Barrier)
- Address (Memory location accessed by the instruction)

- Data (Value read by loads or written by stores)
- Processor (A numerical identifier of the core that issued this instruction)

Of course, traces of programs can have other types of instructions, such as floating-point arithmetic (FADD, FSUB, FMUL, FDIV, etc.); however, these instructions do not access memory and can safely be ignored for the consistency verification process.

Edges of the graph represent *memory ordering* between dynamic instructions. That is, an edge from instruction u to instruction v signifies that instruction u accessed memory before instruction v. Note that memory order is distinct from instruction execution order because in-flight reordering is allowed by the ARM architecture. Indeed, an edge from u to v in the same processor does not imply that u preceded v in the instruction stream.

8.1.2 TSOtool Workflow

Figure 20 illustrates the consistency verification process employed by the work from Hangal *et al.* on TSOtool. The process begins with a randomly generated test program. Test programs can be generated using parameters such as the total number of instructions per core, the number of unique store locations, the ratio of loads to stores, and the types of memory instructions to target various subsets of the memory system. The generated test program is carefully constructed such that each store in the graph writes a *unique value* to memory [121]. The uniqueness of store values provides a trivial mapping of a load to the store that wrote its data, which simplifies the algorithms for analyzing the ensuing graph. Since the data written and read from memory is independent from the behavior of the protocol, using unique store data does not limit the diversity of test cases that can be generated.

Once the test program is generated, it is executed (on a simulator, RTL, or silicon) to obtain the actual data values observed by each load instruction. This information, combined with the rules of the underlying architectural and consistency models allows one to create the initial directed graph for analysis. This graph comprises two classes of edges:

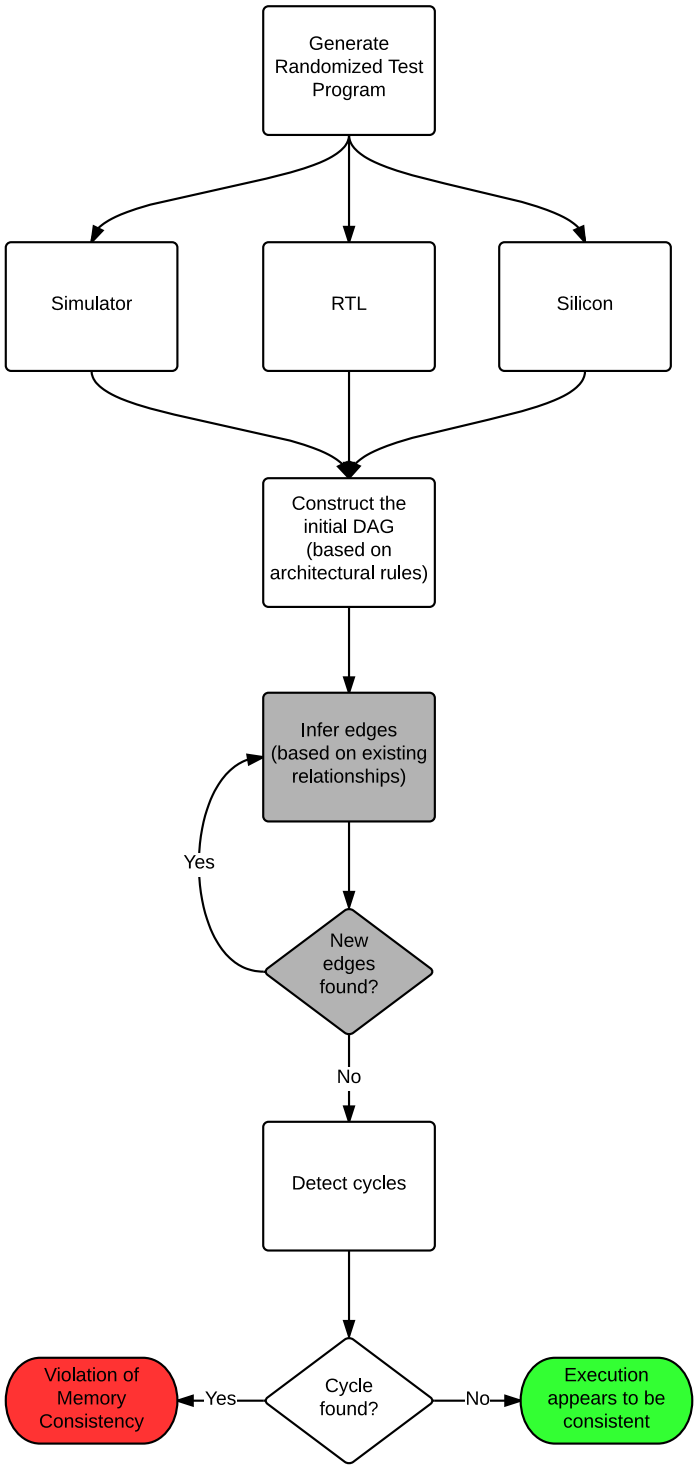


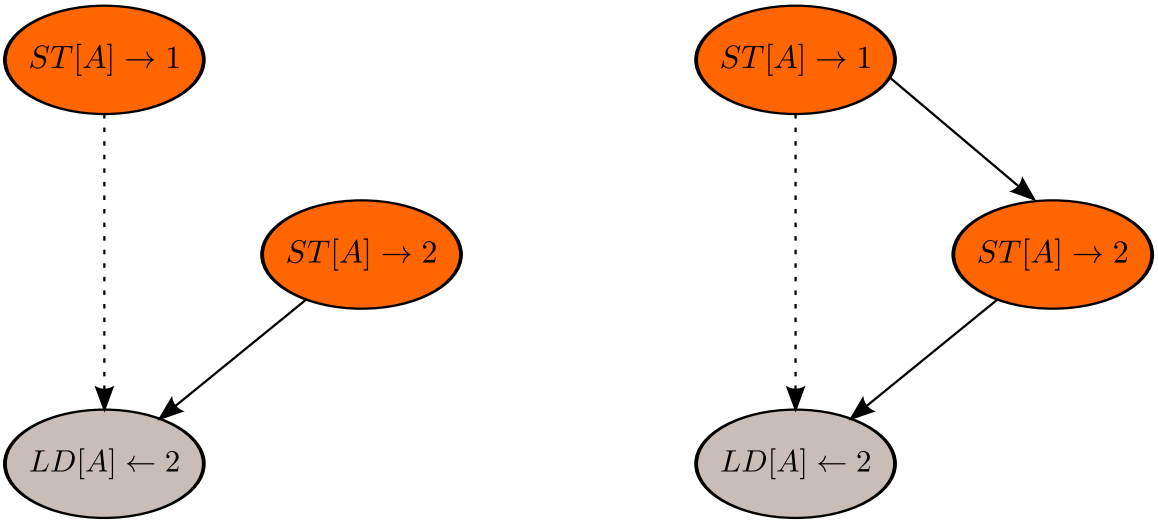
Figure 20: Design flow for memory consistency verification

- *Static* edges. These relationships are enforced by the architecture in the presence of data hazards. For instance, the ARM architecture specifies that operations issued prior to a memory barrier must execute before operations after the memory barrier. The architecture also prevents reordering of loads and stores in the presence of data hazards.

- *Observed* edges. These relationships are enforced by the data read by load instructions during a particular execution of the test program. For example, if a load L on processor p_0 reads the (globally unique) value x from address A and we know that store S on processor p_1 wrote the value of x to A, we can add a directed edge from S to L in the graph, because the consistency model requires that loads read data from the store that most recently wrote to memory.

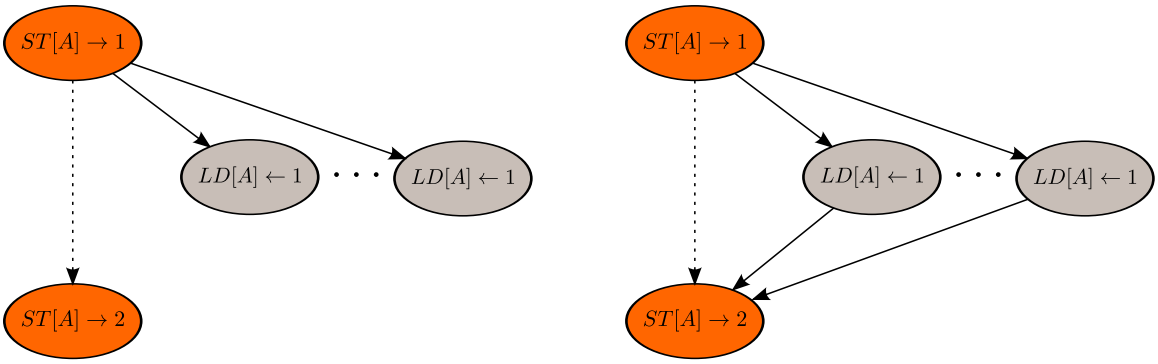
Once the initial graph is constructed, the existing relationships (edges) in the graph can be used to infer additional relationships according to the consistency model. Any such new edges may lead to further relationship inferences, and edges are inferred iteratively until the graph has reached a fixed point. At this point, the graph is checked for cycles in linear time using a technique known as trimming [21]. If the graph contains one or more cycles then we are certain that the consistency model was violated. If the graph has no cycles the execution of the program *appears* to be consistent. Consistency is not guaranteed because these static and observed relationships are not complete: there are further (mutually exclusive) sets of *plausible* relationships that could be established in accordance with the memory model to provide a total ordering of memory instructions and thus a perfectly accurate verification. However, determining whether there exist any such plausible sets that do not induce dependence cycles is NP-complete [126]. For more information regarding static, observed, and inferred edges, we refer the reader to [127].

This iterative process corresponds to the gray boxes in the center of Figure 20, which represent a substantial portion of the overall consistency verification process and thus our focus for optimization and parallelization.



(a) Before (b) After

Figure 21: Example of a Rule 6 inferred edge insertion



(a) Before (b) After

Figure 22: Example of Rule 7 inferred edge insertions

8.1.3 Inferred Edge Insertions

There are two types of inferred edge insertions made by TSOtool, referred to as rule 6 and 7 insertions[1] (using the TSOtool notation [80] and shown in Figures 21 and 22, respectively). Notationally, $ST[A] \rightarrow 1$ means that this instruction wrote the value 1 to location A. Similarly, $LD[A] \leftarrow 2$ means that this instruction read the value 2 from location A. The edges drawn with dotted lines in the Figures denote reachability, meaning that the head of the edge can reach the tail of the edge either directly or transitively. In contrast, the edges

[1] Rules 1-3 cover static edges and rules 4-5 cover observed edges.

drawn with solid lines denote the stronger notion of direct neighbors in the graph.

An example of a Rule 6 insertion is shown in Figure 21. We can see from Figure 21a that the store writing the value 1 can reach a load to the same address that reads the value 2. Since the load reads a different value than the store that can reach it and since store values are unique, there must be another store that writes 2 to location *A* before the load occurs. Furthermore, this store must have accessed memory after the store that wrote 1 because otherwise the load would have read the value 1. Therefore, we can insert an edge from the store that wrote 1 to the store that wrote 2, as shown in Figure 21b.

Figure 22 shows an example of Rule 7 edge insertions. In this case, a series of loads read the value 1, as shown in Figure 22a. Since store values are unique, these loads must all be reading a value written by the same store. If that store can reach another store in the graph that accesses the same location in memory (and, as it must by design, writes different data), we know that the series of loads must all precede the later store since otherwise the loads would have read the value from that store instead (i.e., the value 2). Hence, we can insert edges from each of these loads to the later store, as shown in Figure 22b.

8.2 Sequential Methodology

This section describes several approaches for inferring edges from a constraint graph to solve the memory consistency verification problem. We give an overview of the algorithm used by NVIDIA's application of TSOtool to verify the memory consistency of ARM processors (which have a weaker memory model than TSO). Next, we explain several key performance optimizations to TSOtool [124] that we leverage for our parallel implementation.

8.2.1 Initial Algorithm

Algorithm 20 shows a straightforward approach for an iteration of inferring edges. This process is repeated iteratively until a fixed-point is reached. The outermost loop iterates through each store vertex S in the graph. The inner loop on Line 2 iterates through every

Algorithm 20: Simple Sequential Approach for Inferring Edges

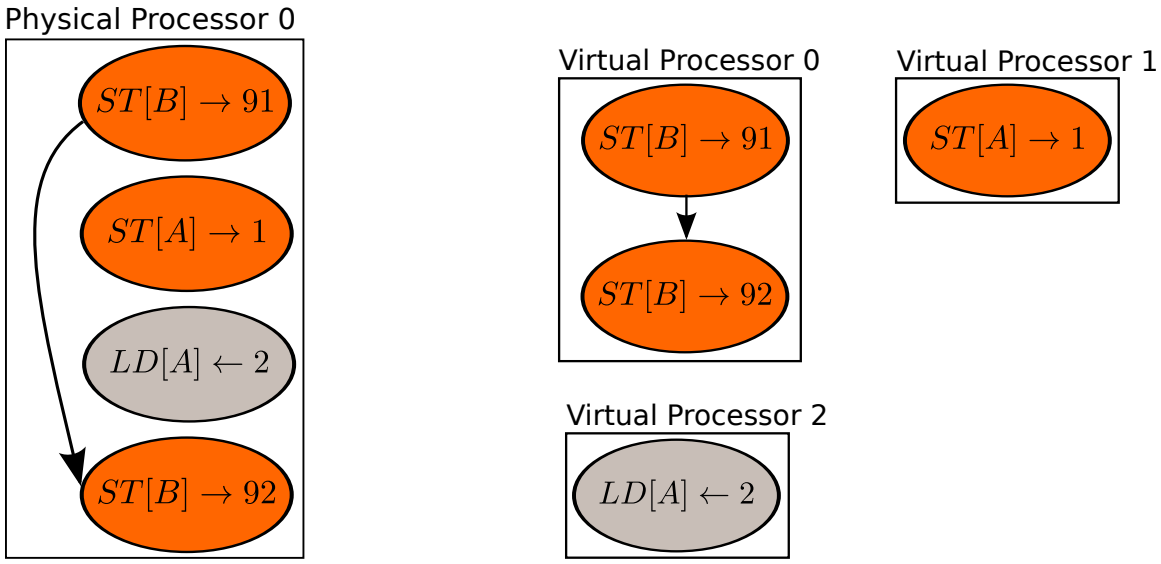
```
1 for {S ∈ V | S.type = ST} do
2     for {X ∈ V | S ≤ X} do
3         if S.location = X.location then
4             if X.type = LD ∧ S.data ≠ X.data then
5                 //Add Rule 6 edge from S to the parent store of X
6             else if X.type = ST then
7                 for {L ∈ V | S.data = L.data} do
8                     //Add Rule 7 edge from L to X
```

reachable vertex from S. Here we use the notation $S \leq X$ to represent that instruction S comes before instruction X in memory order, which is equivalent to saying that X is reachable from S in the graph.

The remaining lines in Algorithm 20 logically enforce the inferred edge rules depicted by Figures 21 and 22. The loop in Line 7 essentially represents the set of loads L that read data from S. The complexity of finding these vertices can be reduced since we can explicitly map each store to its respective child loads once the initial graph is constructed, as this information is constant throughout the execution of the program. Overall, the time complexity of one iteration of Algorithm 20 is $O(n^3)$, assuming that edges can be inserted into the graph in $O(1)$ time.

8.2.2 Virtual Processors and Reverse Vector Time Clocks

Manovit and Hangal develop a more efficient algorithm that leverages transitivity via the use of *virtual processors* and *Reverse Time Vector Clocks* (RTVCs) [124]. The authors discovered that the set of instructions from each physical processor can be grouped into subsets of instructions that belong to virtual processors (vprocs) where each virtual processor is sequentially consistent (and thus have equivalent program order and memory order). Figure 23 shows an example of how a physical processor can be split into sequentially consistent virtual processors. This process depends on the memory model being targeted; for the ARM processors considered in this study, instructions on a physical processor are



(a) Physical (b) Virtual

Figure 23: Splitting of a physical processor in sequentially consistent virtual processors

grouped by their memory location and instruction type. Since instructions that access different locations in memory can be freely reordered by the hardware, they must be assigned to different virtual processors. Although the instructions belonging to vprocs 1 and 2 in Figure 23 access the same memory location A they must also belong to separate virtual processors because the view in which these instructions are executed from *other physical processors* can be out of order.

A consequence of this grouping is that if a vertex S has an outgoing edge to an instruction X in virtual processor p then S implicitly precedes all of the successors of X that are also in p. Thus, a given instruction only needs to inspect its outgoing edges to its earliest successors in each virtual processor [124]. This bounds the number of reachable vertices to be inspected by each store by p, the number of virtual processors, rather than n, the total number of vertices. The data structure that points from each vertex to its earliest successors in each vproc is referred to as a Reverse Time Vector Clock (RTVC) [124], named after the popular approach for partially ordering events in distributed computing [130, 131].

Algorithm 21 shows how the use of virtual processors and reverse time vector clocks

Algorithm 21: Optimized Sequential Approach for Inferring Edges

```
1  for {S ∈ V | S.type = ST} do
2      for {X ∈ S.rtvc[P], P ∈ p} do
3          while X ≠ vprocs[P].end do
4              if X.type = LD ∧ S.data ≠ X.data ∧ S.location = X.location then
5                  //Add Rule 6 edge from S to the parent store of X
6                  update_RTVCs()
7                  break //Move on to next vrpoc
8              else if X.type = ST ∧ S.location = X.location then
9                  for {L ∈ V | S.data = L.data} do
10                     //Add Rule 7 edge from L to X
11                 update_RTVCs()
12                 break //Move on to next vproc
13             else
14                 X ← X.next
```

can significantly reduce the complexity required to infer edges. The RTVC of an instruction S is denoted as $S.rtvc$ and the earliest successor of S to vproc P is denoted as $S.rtvc[P]$. The algorithm simply finds the earliest successors of each store to each vproc for which edges can be inferred, given a test program containing p vprocs. These techniques reduce the complexity of inferring edges from $O(n^3)$ to $O(n^2p^2d_{max})$, where d_{max} is the maximum degree of any vertex $v \in V$. Since $p^2d_{max} < n$ these changes have led to order of magnitude improvements in execution time over the approach outlined in Algorithm 20 [124].

The recomputation of RTVCs ($update\_RTVCs()$) in Lines 6 and 11 of Algorithm 21 deserves separate attention. Algorithm 22 shows the details of this function. Line 1 returns the topological sort of the input graph G in reverse order. This operation is easily completed in linear time [132]. The reverse order of the topological sort is necessary to execute the iterations of the loop on Line 2 in the proper order. The loop on Line 4 looks at the direct neighbors of U. If a given neighbor V of U belongs to a vproc that has yet to be seen from the perspective of U, then it is the earliest successor (so far) from u to that vproc (Line 7). Otherwise, if V belongs to a vproc that U already has an entry for, the program ordering

Algorithm 22: Function for Updating RTVCs

```
1  topo ← reverse(get_topological_sort(G))
2  for {U ∈ topo} do
3      U.rtvc[i] ← ∞, ∀i ∈ p
4      for {V ∈ U.adjacency_list} do
5          W ← U.rtvc[V.vproc]
6          if W = ∞ then
7              U.rtvc[V.vproc] ← V
8          else if V.program_order < W.program_order then
9              U.rtvc[V.vproc] ← V
10         //Now check transitive edges through V
11         for {P ∈ p} do
12             if V.rtvc[P] ≠ ∞ then
13                 if U.rtvc[P] = ∞ then
14                     U.rtvc[P] ← V.rtvc[P]
15                 else if V.rtvc[P].program_order < U.rtvc[P].program_order then
16                     U.rtvc[P] ← V.rtvc[P]
```

of that entry and V can be compared (Line 8) to determine which successor is earlier in memory order (recall that instructions belonging to the same vproc are ordered in memory as they are in the program). The loop on Line 11 looks transitively through the RTVC of V to update the RTVC values of U.

8.3 Facilitating Parallelism

Despite the performance gains seen by the algorithm presented in the previous section, large tests of interest still take days to execute on server class machines. A natural way to elicit further performance gains is to parallelize the algorithm and run it on multi-core CPUs or GPU accelerators. However, Algorithm 21 isn't trivially parallelized. Every time an edge is inserted, the RTVCs of the head of the edge and its ancestors are updated. In general, updating RTVCs requires $O(npd_{max})$ time per edge insertion. There can be up to $O(n^2)$ edge insertions in the worst case, although the number of added edges is typically a small multiple of n. Regardless, the time spent updating RTVCs is significant and these

updates are a barrier to parallelism since the iterations of the inner loops of Algorithm 21 are dependent on the RTVC values.

Algorithm 23: Parallel-friendly Approach for Inferring Edges

```
1  for {S ∈ V | S.type = ST} do
2      for {X ∈ S.rtvc[P], P ∈ p} do
3          while X ≠ vprocs[P].end do
4              if X.type = LD ∧ S.data ≠ X.data ∧ S.location = X.location then
5                  //Add Rule 6 edge from S to the parent store of X
6              else if X.type = ST ∧ S.location = X.location then
7                  for {L ∈ V | S.data = L.data} do
8                      //Add Rule 7 edge from L to X
9              else
10                 X ← X.next
11 update_RTVCs()
```

To enable parallelism we propose a simple alteration to Algorithm 21: reduce the frequency of RTVC updates from once per edge insertion to once per iteration of inferring edges. Algorithm 23 shows this alteration. This change allows for the iterations of the for loops on Lines 1 and 2 to be safely executed independently in parallel at the potential cost of some unnecessary work in the form of edge insertions that provide no information with respect to the memory order of instructions. Even though the graph changes as the algorithm progresses, the work required by an iteration of Algorithm 23 only depends on the RTVC values and not the current state of the graph. When the RTVC values are updated at the end of an iteration of inferring edges, the edges found during the iteration can be inserted into the graph in one batched operation and then the new RTVC values can be derived from the updated graph.

Lazily updating the RTVCs instead of greedily updating them can result in situations where reachable vertices from each store are checked when such a check isn't strictly necessary, as is done in Algorithm 20. However, our evaluation demonstrates that updating RTVCs less frequently is well worth the cost of the extra work.

To show that this strategy maintains the correctness of the approaches outlined in the previous algorithms, it will suffice to show that our method is both sound and "as complete." Let G_i be the graph obtained after i iterations of Algorithm 20 and H_i be the graph after i iterations of Algorithm 23. Let I be the total number of iterations, noting that this value may be different for Algorithms 20 and 23. It follows that G_I and H_I are the resultant graphs after these algorithms terminate.

Theorem 1. *Soundness. All edges inserted by Algorithm 23 represent valid memory orderings of instructions.*

Proof. To show that Theorem 1 is valid, we note that even if RTVC values are "stale," they always point to a successor of S. Since Algorithm 20 iterates through *all* successors of S for inferring edges, any edges that are inserted regardless of whether or not RTVCs are consistent with the current state of the graph must be valid. Any RTVCs that are stale will be updated for the next iteration such that the earliest successors from each store to each vproc will always be checked before the algorithm terminates. Since the Algorithms 20 and 21 have been shown to only insert valid edges [80,124] and since our algorithm only inserts edges that either of these algorithms would insert, our algorithm must also only insert valid edges. □

Theorem 2. *Completeness parity. If G_I contains a cycle then H_I must also contain a cycle.*

Proof. To satisfy Theorem 2, we assume for the purpose of contradiction that H_I has no cycle when G_I has a cycle. This result implies that we have neglected to insert some edge e that created the cycle in G_I. However, Algorithm 23 ensures that the earliest successor from each store vertex to each vproc is checked for the application of rule 6 and 7 edges. Since it was shown that such checks provide the same information as checking all successors from each store [124], e must have been found, else it does not exist. Since e was not found, it must not exist, contradicting our initial assumption. □

In addition to facilitating parallelism, Algorithm 23 performs less work to update RTVCs. If k edges are inserted into the graph, Algorithm 21 requires $O(k)$ RTVC updates. In contrast, Algorithm 23 only requires $O(i)$ RTVC updates for i iterations because the number of RTVC updates scales with the number of iterations rather than the number of edge insertions. Since $k = O(n^2)$ in the worst case (and is $O(n)$ in practice) whereas $i \leq 10$ for all of the test cases used for this study, Algorithm 23 requires significantly less overhead for RTVC updates than Algorithm 21. Overall, Algorithm 23 reduces the work complexity of inferring edges from $O(n^2p^2d_{max})$ to $O(n^2p)$ per iteration.

8.4 Parallel Methodology

The approach to verifying memory consistency described in the previous section is not the first attempt to parallelize this class of algorithms. Roy *et al.* present a parallel implementation of TSOtool that targeted the Intel IA-32 and Itanium architectures [133]. Their approach cannot utilize virtual processors and the RTVC-based optimizations described in [124] because of complications arising from using specific memory types on the IA-32 and Itanium architectures.

Although these complications encouraged the design of a more general approach, the algorithms from Roy *et al.* have a few significant weaknesses. Firstly, they choose to store the graph as an adjacency matrix, requiring $O(n^2)$ space despite the fact that constraint graphs are typically quite sparse. This decision prohibited scalabity to graphs larger than $10,000$ vertices in their experiments. Secondly, it is unclear if the tests performed by Roy *et al.* are scalable to large thread counts or portable to graphs with differing characteristics, such as the number of accessed memory locations or the ratio of loads to stores.

In the remainder of this section we discuss two parallel implementations of Algorithm 23: one in OpenMP for multi-core CPUs and the other in CUDA for GPU accelerators.

8.4.1 OpenMP

Considering that shared-memory systems using OpenMP tend to have a limited number of threads, decomposing problems such that each thread has a sufficient amount of work is less challenging than on systems with thousands of concurrent threads, such as NVIDIA's GPUs or Intel's Xeon Phi coprocessors. Since we know in advance that the number of threads is small, we provide each thread with its own storage space for collecting newly inserted edges to reduce communication overhead. Threads are assigned to iterations of the for loop in Line 1 of Algorithm 23, which can independently traverse the RTVCs from their assigned store vertex and add edges to their local lists. Once the entire iteration is complete these thread-specific lists are trivially reduced into one global list, which is used to update the graph.

Algorithm 24: OpenMP Approach for Inferring Edges

```
1  added_edges ← vector(num_threads())
2  for {S ∈ V | S.type = ST} do in parallel
3      for {X ∈ S.rtvc[P], P ∈ p} do
4          while X ≠ vprocs[P].end do
5              if X.type = LD ∧ S.data ≠ X.data ∧ S.location = X.location then
6                  added_edges[get_id()].insert(S → X)
7              else if X.type = ST ∧ S.location = X.location then
8                  for {L ∈ V | S.data = L.data} do
9                      added_edges[get_id()].insert(L → X)
10             else
11                 X ← X.next
12 foreach {Partition t ∈ num_threads()} do
13     foreach {Edge e ∈ added_edges[t]} do
14         new_edges.insert(e)
15 update_RTVCs()
```

Algorithm 24 shows the details of this approach. The vector *added_edges* of length *num_threads*(), the number of OpenMP threads, allows each thread to concurrently find edges without communication or race conditions. The for loop on Line 12 sequentially

accumulates the results collected this way into one data structure so that one large update to the graph can be made instead of *num_threads*() (smaller) updates. Although we had the option of utilizing finer granularities of parallelism, using the coarsest level of granularity maximizes independent work among threads and minimizes the OpenMP overhead of creating and destroying threads.

8.4.2 CUDA

Our initial approach to parallelizing Algorithm 23 using CUDA involved a slightly more complicated thread decomposition than our OpenMP approach. We initially assigned thread blocks (groups of threads) to each store processed by the outermost loop on Line 1 of Algorithm 23. The threads within each block were assigned to inspect each vproc from their respective store as seen on Line 2 of Algorithm 23. This approach achieved limited processor utilization because the number of vprocs is small relative to the number of threads per block, which should be a multiple of the warp size (currently 32 threads on NVIDIA hardware). Additionally, the work done by each thread in this manner is fairly uneven. At one end of the spectrum, a thread may find that its store has a null RTVC value for the vproc that it is looking at and thus, the thread has no work to complete. In contrast, another thread may simultaneously find that the store does have an RTVC entry to this vproc but the earliest successor from the store to the vproc is much later than the initial entry. In this latter case the thread must traverse through the vproc and possibly insert edges from each load that reads from the store to this successor.

It turns out that simply taking advantage of the large amount of coarse-grained parallelism through the number of stores ($O(n)$) is a better approach. This approach more efficiently utilizes the processor because threads are constantly kept busy by processing independent store vertices rather than waiting for the critical thread in a given block to finish traversing its vproc and adding edges. Using our initial approach, if the number of vprocs modulo 32 (the warp size) is not 0 then threads will have an unequal number of vprocs to inspect. In the worst case, one thread has one more vproc than all of the others, meaning

that the remaining threads in the block will all idle while the one thread with additional work inspects its additional vproc. Using a coarser approach eliminates this issue because each thread processes all vprocs from a given store. A load imbalance may still exist in the number of stores to be processed per thread; however, the small number of Streaming Multiprocessors (SMs) on the GPU implies that the same HW lanes will sequentially process many stores, making this "off by one" load imbalance insignificant.

Algorithm 25: CUDA Approach for Inferring Edges

```
1  for {S ∈ V | S.type = ST} do in parallel
2      for {X ∈ S.rtvc[P], P ∈ p} do
3          while X ≠ vprocs[P].end do
4              if X.type = LD ∧ S.data ≠ X.data ∧ S.location = X.location then
5                  t ← atomicAdd(&edge_ptr, 1) new_edges[t].insert(S → X)
6              else if X.type = ST ∧ S.location = X.location then
7                  for {L ∈ V | S.data = L.data} do
8                      t ← atomicAdd(&edge_ptr, 1) new_edges[t].insert(L → X)
9              else
10                 X ← X.next
11 update_RTVCs()
```

Algorithm 25 shows how we alter our parallel implementation for a GPU architecture. Having separate subarrays for each thread as was done in Algorithm 24 is no longer practical because of the large (and tunable) number of threads offered by the GPU. We instead use one array and use atomic operations (Lines 5 and 8) to ensure that threads write to unique locations in memory. In practice we calculate the number of edges to be inserted by the loop on Line 7 and perform one *atomicAdd* rather than one *atomicAdd* for each inserted edge. Since the logic of finding which edges to add is the bottleneck of this algorithm we can easily update the graph (and RTVCs) on the CPU between iterations of Algorithm 25.

8.5 Results

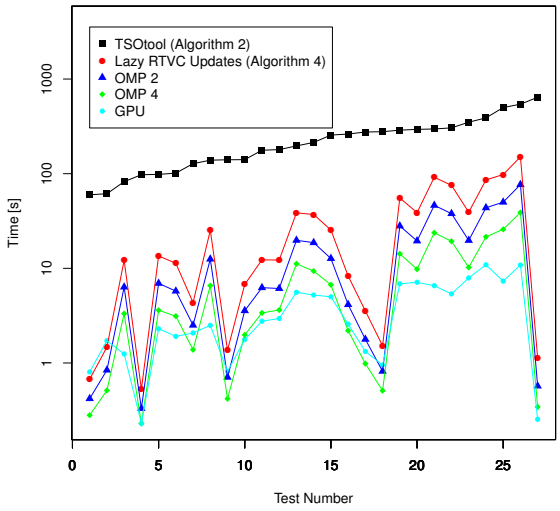
8.5.1 Experimental Setup

All experiments were run on a system with an Intel Core i7-2600K CPU and an NVIDIA GeForce GTX Titan GPU. The Intel Core i7-2600K has four cores, each of which run at 3.4 GHz, an 8 MB cache, and 16 GB of DRAM. The NVIDIA GeForce GTX Titan has a base clock that runs at 837 MHz, 6 GB of GDDR5 memory, a peak theoretical memory bandwidth of 288.4 GB/s, and is a compute capability 3.5 ("Kepler") GPU.

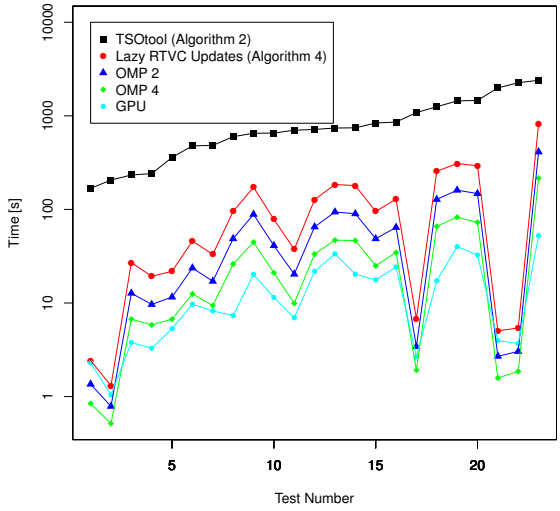
Sequential and OpenMP code was written in C++ and compiled with version 4.5.3 of the g++ compiler. CUDA code was compiled with the nvcc compiler and the CUDA 6.0 toolkit. We compare our approaches to an adaptation of TSOtool used to verify ARM processors. The system we use for testing contains four ARM Cortex-A57 cores. The Cortex-A57 microarchitecture implements the ARMv8-A 64-bit instruction set and has an out-of-order superscalar pipeline. The graphs we use for experimentation represent real traces used to find bugs in the implementation of the memory model (or bugs in the memory model itself). The graphs span sizes ranging from $n = 2^{18}$ to $n = 2^{22}$ vertices, with each vertex representing an instruction from one of the four processor cores. Each core issues the same number of instructions. The precise number of edges that each graph initially contains varies, but is fairly close to n. Hence, these graphs represent a particularly sparse, high-diameter, and low-degree network structure. We test a number of graphs of each size. These graphs vary in their proportion of load, store, and barrier instructions; number of virtual processors; and number of instruction dependencies.

8.5.2 Experimental Results

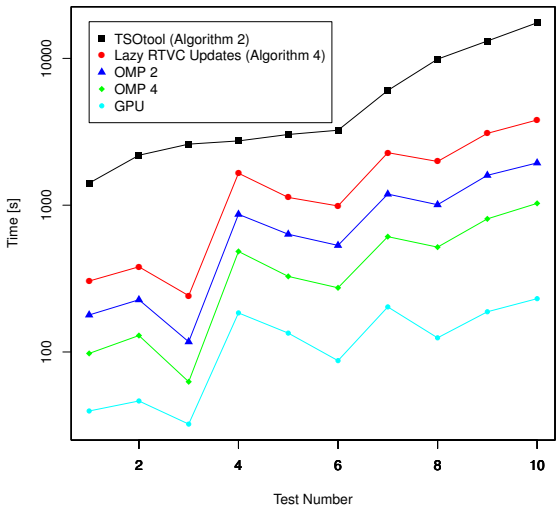
Figure 24 shows the performance characteristics of TSOtool and our improvements over this baseline. The data shown in these figures represent only the time spent adding edges to the graph; however, we will show that this time represents a vast majority of the overall execution time, and hence was our focus for optimization. Figure 24a shows tests that all have 128K instructions per core, or a total of $n = 2^{19}$ instructions across all four cores.



(a) 128K Instructions per Core



(b) 256K Instructions per Core



(c) 512K Instructions per Core

Figure 24: Performance results for various test sizes

Similarly, Figures 24b (and 24c) show tests that all have 256K (512K) instructions per core, or a total of $n = 2^{20}$ ($n = 2^{21}$) instructions. The tests are independent, but are sorted from fastest to slowest (for the TSOtool baseline) for convenience. We compare four of our approaches to the TSOtool baseline:

1. A sequential approach that minimizes updates to RTVCs (Algorithm 23)
2. An OpenMP implementation using 2 threads (OMP 2, Algorithm 24)
3. An OpenMP implementation using 4 threads (OMP 4, Algorithm 24)
4. A GPU implementation (Algorithm 25)

Note that by inspection of the (logarithmic) y-axis of Figures 24b and 24c one can see that for a graph that is just twice as large, experiments can take significantly longer to run. It is evident from Figure 24 that TSOtool spends excessive time updating RTVCs. Our alternative sequential method that lazily, rather than eagerly, updates RTVCs (as shown in Algorithm 23) shows substantial improvements over this baseline. Furthermore, since our algorithm facilitates parallelism, we attain additional performance improvements by using OpenMP and CUDA. In the more extreme cases, our GPU implementation is orders of magnitude faster than TSOtool.

It is interesting to note that although the results for TSOtool have been plotted in order of increasing execution time, our corresponding implementations do not necessarily share this behavior. For instance, the second slowest TSOtool test in Figure 24b executes much faster than the slowest TSOtool test for our implementations. This peculiarity is explained by the fact that this particular test has a larger portion of store instructions (79%) than the other tests of this size. A larger number of store instructions leads to more executions of the outer for loop of Algorithm 21 in comparison to other tests which also leads to a relatively greater number of calls to $update\_RTVCs()$. Since our approach in Algorithm 23 improves upon the previous approach in Algorithm 21 precisely by calling $update\_RTVCs()$ less frequently it makes sense that our approach would perform especially well for this particular

Table 20: Speedup over TSOtool for our sequential and parallel implementations of inferring edges

Inst. per Core	64K	128K	256K	512K	1M
Num. of tests	27	27	23	10	2
Alg. 23	15.09x	16.41x	14.51x	4.01x	3.08x
OMP 2	29.31x	31.49x	27.98x	7.52x	5.70x
OMP 4	53.45x	57.34x	51.68x	14.19x	10.39x
GPU	57.90x	76.98x	72.32x	42.90x	45.16x

Table 21: Parallel Speedups over Algorithm 23

Inst. per Core	64K	128K	256K	512K	1M
Num. of tests	27	27	23	10	2
OMP 2	1.94x	1.92x	1.93x	1.88x	1.85x
OMP 4	3.54x	3.49x	3.56x	3.54x	3.37x
GPU	3.84x	4.69x	4.98x	10.70x	14.66x

test.

Table 20 shows the geometric mean speedup of our various approaches over TSOtool for each test size. Note that the number of tests decreases with test size due to industrial time constraints when using TSOtool, providing motivation for our efforts. We can see that our reduction in the number of RTVC updates gives us at least a 3x speedup over TSOtool sequentially. Furthermore, since our methodology facilitates parallelism, we see the additional benefit of parallelism, as shown in Table 21. Table 21 shows the precise performance gains for inferring edges in parallel. The speedups for parallel methods in Table 20 show total speedup over the TSOtool baseline, which includes the speedup of simply using the more efficient sequential algorithm as well as parallel performance benefits. Table 21 extracts the parallel speedups over our more efficient sequential approach (Algorithm 23) to convey the benefits of parallelization alone. We can see that the OpenMP implementations approximately achieve 1.9x and 3.5x speedups using 2 threads and 4 threads, respectively, regardless of problem size. Our GPU implementation consistently does better than the OpenMP implementation; however, it doesn't perform substantially better than

Table 22: Application speedup over TSOtool for our sequential and parallel implementations

Inst. per Core	64K	128K	256K	512K	1M
Num. of tests	27	27	23	10	2
Alg. 4	5.64x	5.31x	6.30x	3.68x	3.05x
OMP 2	7.62x	7.12x	9.05x	6.41x	5.58x
OMP 4	9.43x	8.90x	12.13x	10.81x	9.97x
GPU	10.79x	10.76x	15.47x	24.55x	37.64x

the OpenMP implementation until the problem size is sufficiently large. Compared to our sequential approach, the GPU approach achieves more than an order of magnitude speedup for graphs with 512K instructions per core or greater.

Recall from Figure 20 that inferring edges is just one portion of the overall design flow for the memory consistency verification problem. Thus, we need to show that our efforts in improving the performance of inferring edges also improves overall application performance, else our efforts were not properly focused. Table 22 shows the overall application speedup of our sequential and parallel approaches over TSOtool. These speedups quantify the additional throughput one can achieve in terms of the number of tests run by using our approaches. For instance, one can run approximately 37 times as many tests with 1M instructions per core using our GPU implementation compared to what is done today in the same amount of time using TSOtool. This increase in throughput is important because it allows for greater coverage in testing. Running additional tests allows one to check for a greater variety of errors in the memory subsystem.

These speedups are also substantial in terms of absolute time and performance. One of the larger tests we experimented with required over nine hours of total application execution time using TSOtool. Using our GPU approach, we were able to finish the same test in under ten minutes. As a metric of absolute performance, we measured the GPU memory throughput of our larger tests to average 28.19 GB/s and reach a peak of 35.15

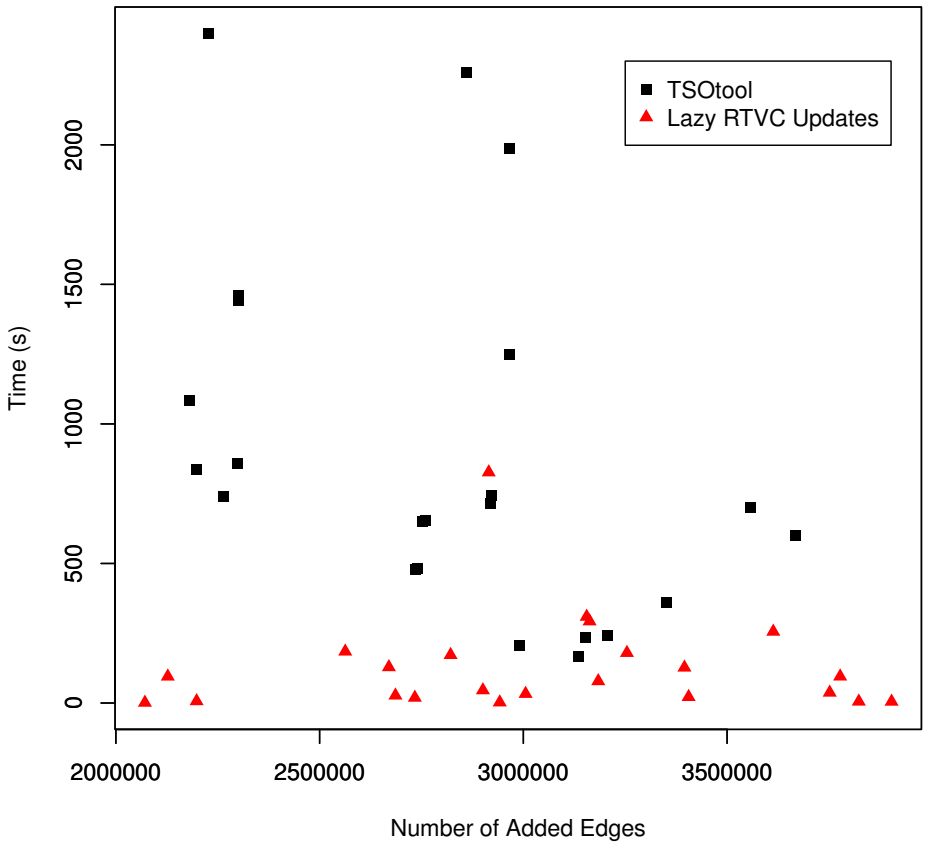


Figure 25: Scatter plot of edges added and performance

GB/s, showing that our implementation, although simplistic, efficiently utilizes the processor. Considering the extreme sparsity and irregular structure of the program execution graphs that we tested, achieving a significant percentage of the peak memory bandwidth of the processor is challenging [14].

Figure 25 provides some additional performance insights. We compare both sequential methods of updating RTVCs, eager (TSOtool, Algorithm 21) and lazy (Algorithm 23), in terms of performance and the number of inferred edges. A consequence of using Algorithm 23 and infrequently updating RTVCs is that stale RTVC values can lead to unnecessarily inferred edges. However, in some cases we can see that using Algorithm 23 actually results in *fewer* inferred edges. This result can occur when a store's RTVC to a certain vproc incrementally moves toward the beginning of the vproc during an iteration of Algorithm 21. In contrast, Algorithm 23 will skip the intermediate locations of the RTVC, thus

Table 23: Metadata regarding the twelve largest test cases

n = |V|	m = |E|	TSOtool Inferred	Iter.	ST/LD/BAR (%)
2,097,963	3,799,254	4,487,224	5	76/24/0
2,098,219	3,686,624	4,411,887	4	79/21/0
1,977,832	4,453,340	5,179,108	5	46/53/1
2,097,741	3,875,831	4,635,852	7	77/23/0
1,936,321	5,109,990	5,236,671	5	44/54/2
2,098,321	2,491,062	4,257,077	6	80/20/0
2,097,809	4,321,793	4,404,753	7	78/21/1
1,871,831	3,660,617	4,861,044	6	44/54/2
2,097,809	4,434,120	4,418,555	5	80/20/0
2,004,180	4,354,887	5,530,123	6	45/54/1
4,195,405	6,934,725	9,338,902	7	76/23/1
4,194,961	7,960,567	8,963,281	6	78/22/0

neglecting to infer any edges at those intermediate locations.

Although it was shown in Section 8.3 that inferring additional edges or neglecting to infer these intermediate edges does not invalidate the program's output, it is reasonable to be concerned about the performance implications of the unnecessary work of inferring additional edges. Figure 25 shows that the amount of execution time for tests run using Algorithm 23 is independent of the number of edges inferred. In fact, the (Pearson product-moment) correlation coefficient between these two vectors is just 0.007, supporting that the data are largely unrelated. Surprisingly, for tests run using Algorithm 21 we actually see a slight *inverse* correlation between execution time and the number of inferred edges: -0.423. It is clear that other characteristics, such as the size of the graph, the frequency of dependencies between instructions, and the distribution of instruction types have a more profound impact on performance. For our largest tests we saw up to 36% additional edges inserted by Algorithm 23 compared to that of Algorithm 21; nevertheless, our speedups still justify the redundant work.

Table 23 shows additional information regarding our twelve largest test cases. The first two columns present the number of vertices and edges for each test case, respectively, before any edges are inferred. It is evident that the initial graphs are substantially sparse, as $m < 3n$. The third column shows the number of edges that are inferred using the algorithm from TSOtool. We place this data side by side with the number of iterations (shown in the fourth column) because these two columns contrast the number of calls to $update\_RTVCs()$ made by TSOtool and our approaches. We can see that using TSOtool, $O(n)$ calls to $update\_RTVCs()$ are made for these test cases, and these excessive calls will only become increasingly detrimental to performance as the graphs tested continue to grow. On the contrary, our approach requires at most seven calls to $update\_RTVCs()$. The speedup achieved by our approach isn't directly proportional to this reduction in the number of updates because each iteration of the algorithm requires searching through the vprocs of each store in the graph, regardless of the number of updates that occur. Although the number of iterations tends to grow with the size of the graph, the rate at which the number of iterations grows is tremendously small. Hence, the number of RTVC updates that we perform scales very well with the size of the graph. Finally, the fifth column of Table 23 breaks down the percentage of store, load, and barrier instructions found within each test. Note that tests with the same initial graph and proportion of ST/LD/BAR instructions can still vary by quite a bit as the number of distinct memory locations and virtual processors may differ.

8.6 Conclusions

This paper discusses several parallel methodologies for verifying the memory consistency of architectures with relaxed memory models. We provide an alternative approach to using reverse time vector clocks that chooses to update this data structure after every iteration of inferring edges rather than after every edge insertion as was done previously. This approach

reduces the work complexity of inferring edges, which we have shown to be the dominating factor in terms of performance for the entire verification process. Additionally, this approach simplifies the parallelization of consistency verification as a direct parallelization of our new approach requires significantly less communication than a direct parallelization of the previous approach. For a set of 89 tests in use at NVIDIA, we achieved geometric mean speedups of 12.74x, 44.95x, and 64.28x over the best existing approach for inferring edges for our sequential, OpenMP, and GPU implementations respectively. For the twelve largest test cases, our GPU implementation was able to achieve an average application speedup of 26.36x, reducing execution time from over nine hours to under ten minutes in one instance.

A number of insights regarding the computation of parallel graph algorithms have appeared in recent literature. Frameworks such as Ligra [82] and Galois [84] alleviate the difficulty of programming graph algorithms on shared memory architectures without sacrificing performance. Heavily optimized GPU implementations of specific algorithms have also been developed, for algorithms such as Breadth-First Search [10], Single-Source Shortest Paths [34], and Betweenness Centrality [1]. This collection of work tends to focus on graph algorithms that are traversal-based; it remains unclear if these insights can be directly applied to non traversal-based algorithms such as the algorithms discussed in this paper. We consider a more general approach to the design of shared memory parallel graph algorithms to be an intriguing area of future work.

CHAPTER 9

CONCLUSION

This thesis provided novel techniques for executing fast, scalable, and energy-efficient graph analytics on a variety of GPU platforms. We present parallel, distributed, hybrid, and on-line approaches to computing Betweenness Centrality, a popular analytic used to find influential vertices. Our approaches are especially useful for high diameter graphs such as road networks, which were typically neglected by prior art. Our single GPU approach performs 2.71× faster than the previously best algorithms and using a cluster of 192 GPUs, our multi-node implementation is capable of exceeding 10 GTEPS.

We also considered streaming (or dynamic) approaches to computing BC on the GPU, which, to our knowledge, had not been previously attempted for any graph algorithm on the GPU. Although the exact speedups are heavily workload dependent, our experiments found that our dynamic approach was up to 100× faster than dynamic approaches on the CPU and 45× faster than static approaches on the GPU.

In addition to obtaining excellent performance, our techniques also require less energy than previous work. We have an 83% average reduction in energy-to-solution when computing BC dynamically rather than statically on the GPU. We have shown that our implementation is effective on small embedded devices such as NVIDIA's Kayla platform and large distributed systems such as the Keeneland Initial Delivery System (KIDS).

We were able to improve upon our techniques by using warp-synchronous programming methods as a result of a better understanding of the underlying characteristics of the GPU architecture. Furthermore, we generalized our approach into a programming paradigm that fits any problem requiring many simultaneous breadth-first searches. Although BC is a great fit for this problem, other classical graph algorithms fit it as well: transitive closures, reachability querying, diameter computations, and more. Using this *multi-search* abstraction led to an average speedup of 7.66× and 5.82× over Galois and

Ligra, two parallel CPU frameworks for graph analysis, for computations of BC. Additionally, we had an average speedup of 3.07× over Gunrock, a GPU graph framework, and 2.24× over our previous, manually tuned implementation for the same computation. Finally, we use a series of hierarchical queues to save 42% time and 62% energy over existing implementations of betweenness centrality from the literature.

Our work on parallel graph algorithms on the GPU has also seen attention in industry. In collaboration with NVIDIA, we used some of the techniques presented in this thesis to quickly verify the memory consistency of architectures with relaxed memory models. For a set of 89 tests we achieved mean speedups of 12.74×, 44.95×, and 64.28× over the best existing approach for iteratively inferring memory ordering dependencies for our improved sequential, OpenMP, and GPU implementations respectively. For the twelve largest test cases, our GPU implementation was able to achieve an average application speedup of 26.36×, reducing execution time from over nine hours to under ten minutes in one instance.

These contributions lead to many fascinating avenues for future research. Heterogeneous programming models are still fairly primitive, and finding appropriate levels of abstraction to maximize developer productivity as well as processor utilization is a particularly challenging task. Designing new systems that are more easily programmed, provide better memory bandwidth and opportunities for high throughput distributed computing, and aid in the design verification process is pivotal to efficiently using the time of experts in the parallel computing industry.

REFERENCES

[1] A. McLaughlin and D. A. Bader, "Scalable and High Performance Betweenness Centrality on the GPU," in *Proceedings of the 26th ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis (SC)*, 2014. Best student paper finalist.

[2] A. McLaughlin and D. A. Bader, "Revisiting Edge and Node Parallelism for Dynamic GPU Graph Analytics," in *Eighth Workshop on Multithreaded Architectures and Applications (MTAAP)*, 2014.

[3] A. McLaughlin, J. Riedy, and D. A. Bader, "Optimizing Energy Consumption and Parallel Performance for Static and Dynamic Betweenness Centrality using GPUs," in *Eighteenth IEEE High Performance Extreme Computing Conference (HPEC)*, 2014. Rising stars session.

[4] A. McLaughlin and D. A. Bader, "Fast Execution of Simultaneous Breadth-First Searches of Sparse Graphs," in *Twentieth IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2015. (in submission).

[5] A. McLaughlin, D. Merrill, M. Garland, and D. A. Bader, "Parallel Methods for Verifying the Consistency of Weakly-Ordered Architectures," in *Twenty-fourth ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[6] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229 – 234, 1985.

[7] J. Hoberock and N. Bell, "Thrust: A Parallel Template Library," *Online at http://thrust. googlecode. com*, vol. 42, p. 43, 2010.

[8] D. Merrill, "CUDA Unbound." 2013 (accessed October 21, 2014).

[9] S. Beamer, K. Asanović, and D. Patterson, "Direction-Optimizing Breadth-First Search," in *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 12:1–12:10, 2012.

[10] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 117–128, ACM, 2012.

[11] L. Luo, M. Wong, and W.-m. Hwu, "An Effective GPU Implementation of Breadth-First Search," in *Proceedings of the 47th Design Automation Conference*, pp. 52–55, ACM, 2010.

[12] P. Harish and P. Narayanan, "Accelerating Large Graph Algorithms on the GPU using CUDA," in *High Performance Computing (HiPC)*, pp. 197–208, Springer, 2007.

[13] S. Xiao and W. chun Feng, "Inter-block GPU Communication via Fast Barrier Synchronization," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–12, April 2010.

[14] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," in *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pp. 267–276, 2011.

[15] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[16] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Computing Strongly Connected Components in Parallel on CUDA," in *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pp. 544–555, May 2011.

[17] J. Leskovec and C. Faloutsos, "Sampling from Large Graphs," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, (New York, NY, USA), pp. 631–636, ACM, 2006.

[18] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[19] S. Hong, N. C. Rodia, and K. Olukotun, "On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 92, ACM, 2013.

[20] L. Fleischer, B. Hendrickson, and A. Pinar, "On Identifying Strongly Connected Components in Parallel," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (London, UK, UK), pp. 505–511, Springer-Verlag, 2000.

[21] W. Mclendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding Strongly Connected Components in Distributed Graphs," *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, pp. 901–910, 2005.

[22] L. C. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, pp. 35–41, 1977.

[23] M. Girvan and M. E. Newman, "Community Structure in Social and Biological Networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[24] S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. C. Wong, "A Novel Application of Parallel Betweenness Centrality to Power Grid Contingency Analysis," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–7, 2010.

[25] E. Bullmore and O. Sporns, "Complex Brain Networks: Graph Theoretical Analysis of Structural and Functional Systems," *Nature Reviews Neuroscience*, vol. 10, no. 3, pp. 186–198, 2009.

[26] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, pp. 345–, June 1962.

[27] U. Brandes, "A Faster Algorithm for Betweenness Centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[28] G. Tan, D. Tu, and N. Sun, "A Parallel Algorithm for Computing Betweenness Centrality," in *International Conference on Parallel Processing (ICPP)*, pp. 340–347, Sept 2009.

[29] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart, "Edge v. Node Parallelism for Graph Centrality Metrics," *GPU Computing Gems*, vol. 2, pp. 15–30, 2011.

[30] Z. Shi and B. Zhang, "Fast Network Centrality Analysis using GPUs," *BMC bioinformatics*, vol. 12, no. 1, p. 149, 2011.

[31] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek, "Betweenness Centrality on GPUs and Heterogeneous Architectures," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, (New York, NY, USA), pp. 76–85, ACM, 2013.

[32] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda, "A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–8, May 2009.

[33] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU Implementation of Inclusion-based Points-to Analysis," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 107–116, ACM, 2012.

[34] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single source shortest paths," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, pp. 349–359, May 2014.

[35] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation ," *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012.

[36] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. Farber, and W. N. Reynolds, "Massive Social Network Analysis: Mining Twitter for Social Good," in *IEEE International Conference on Parallel Processing (ICPP)*, pp. 583–593, 2010.

[37] S. Porta, V. Latora, F. Wang, E. Strano, A. Cardillo, S. Scellato, V. Iacoviello, and R. Messora, "Street Centrality and Densities of Retail and Services in Bologna, Italy," *Environment and Planning B: Planning and design*, vol. 36, no. 3, pp. 450–465, 2009.

[38] F. Liljeros, C. R. Edling, L. A. Amaral, H. E. Stanley, and Y. Aberg, "The Web of Human Sexual Contacts," *Nature*, vol. 411, no. 6840, pp. 907–908, 2001.

[39] J. Soman and A. Narang, "Fast Community Detection Algorithm with GPUs and Multicore Architectures," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 568–579, IEEE, 2011.

[40] D. J. Watts and S. H. Strogatz, "Collective Dynamics of Small-World Networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[41] U. Brandes, "On Variants of Shortest-Path Betweenness Centrality and their Generic Computation," *Social Networks*, vol. 30, no. 2, pp. 136–145, 2008.

[42] O. Green and D. A. Bader, "Faster Betweenness Centrality Based on Data Structure Experimentation," *Procedia Computer Science*, vol. 18, no. 0, pp. 399 – 408, 2013. 2013 International Conference on Computational Science.

[43] J. J. Dongarra, H. W. Meuer, and E. Strohmaier, "Top500 Supercomputer Sites," 1994.

[44] "ORNL Debuts Titan Supercomputer." 2012 (accessed April 10, 2014).

[45] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. Çatalyürek, "Regularizing Graph Centrality Computations," *Journal of Parallel and Distributed Computing (JPDC)*, 2014.

[46] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[47] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, eds., *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge*, vol. 588 of *Contemporary Mathematics*, 2013.

[48] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities Based on Ground-Truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, p. 3, ACM, 2012.

[49] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining.," in *SDM*, vol. 4, pp. 442–446, SIAM, 2004.

[50] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and Mobility: User Movement in Location-Based Social Networks," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1082–1090, ACM, 2011.

[51] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a Scalable High Quality Graph Partitioner," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, April 2010.

[52] J. S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community," *Computing in Science & Engineering*, vol. 13, no. 5, pp. 90–95, 2011.

[53] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating Betweenness Centrality," in *Algorithms and models for the web-graph*, pp. 124–137, Springer, 2007.

[54] U. Brandes and C. Pich, "Centrality Estimation in Large Networks," *International Journal of Bifurcation and Chaos*, vol. 17, no. 07, pp. 2303–2318, 2007.

[55] M. Anderson, "Better Benchmarking for Supercomputers," *Spectrum, IEEE*, vol. 48, no. 1, pp. 12–14, 2011.

[56] D. Merrill, M. Garland, and A. Grimshaw, "Policy-based Tuning for Performance Portability and Library Co-Optimization," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–10, IEEE, 2012.

[57] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on gpus," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, (New York, NY, USA), pp. 96–107, ACM, 2013.

[58] H. Jeong, S. P. Mason, A. L. Barabsi, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, pp. 41–42, May 2001.

[59] M. Rubinov and O. Sporns, "Complex network measures of brain connectivity: Uses and interpretations," *NeuroImage*, vol. 52, no. 3, pp. 1059–1069, 2010.

[60] O. Green, R. McColl, and D. A. Bader, "A fast algorithm for streaming betweenness centrality," in *Proceedings of the 2012 ASE/IEEE International Conference on Social Computing (SOCIALCOM)*, (Washington, DC, USA), pp. 11–20, 2012.

[61] M. Harris, "Optimizing Parallel Reduction in CUDA," tech. rep., nVidia, 2008.

[62] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung, "Qube: A quick algorithm for updating betweenness centrality," in *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, (New York, NY, USA), pp. 351–360, ACM, 2012.

[63] M. Kas, M. Wachs, K. M. Carley, and L. R. Carley, "Incremental algorithm for updating betweenness centrality in dynamically growing networks," in *Proceedings of the 5th International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '13.

[64] L. Li, D. Alderson, J. C. Doyle, and W. Willinger, "Towards a theory of scale-free graphs: Definition, properties, and implications," *Internet Mathematics*, vol. 2, no. 4, pp. 431–523, 2005.

[65] D. A. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," 2006.

[66] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *The IEEE High Performance Extreme Computing Conference (HPEC)*, (Waltham, MA), Sept. 2012. Best paper award.

[67] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM Computer Communication Review*, vol. 29, pp. 251–262, ACM, 1999.

[68] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the International Conference on Knowledge Discovery and Data mining (KDD)*, 2003.

[69] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[70] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[71] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili, "Red Fox: An execution environment for relational query processing on GPUs," in *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[72] L. Dematté and D. Prandi, "GPU computing for systems biology," *Briefings in bioinformatics*, vol. 11, no. 3, pp. 323–333, 2010.

[73] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[74] "The Graph500 list." `http://www.graph500.org/results_nov_2013`, Nov. 2013. (accessed on 17 April 2014).

[75] Y. Jia, J. Hoberock, M. Garland, and J. C. Hart, "On the visualization of social and other scale-free networks," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 14, no. 6, pp. 1285–1292, 2008.

[76] P. Crucitti, V. Latora, and S. Porta, "Centrality in networks of urban streets," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 16, no. 1, 2006.

[77] J. P. Bagrow and E. M. Bollt, "Local method for detecting communities," *Phys. Rev. E*, vol. 72, p. 046108, Oct 2005.

[78] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future. concurrency and computation: Practice and experience," *Concurrency and Computation: Practice and Experience*, vol. 15, p. 2003, 2003.

[79] "The Third Green Graph 500 list." `http://green.graph500.org/lists.php`, June 2014. (accessed on 12 July 2014).

[80] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[81] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs.," *OSDI*, vol. 12, no. 1, p. 2, 2012.

[82] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN Notices*, vol. 48, pp. 135–146, ACM, 2013.

[83] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *CoRR*, vol. abs/1501.05387, Jan. 2015.

[84] D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," in *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 456–471, 2013.

[85] J. Kepner, D. A. Bader, A. Buluc, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, Matrices, and the GraphBLAS: Seven Good Reasons," *arXiv preprint arXiv:1504.01039*, 2015.

[86] S. Pallottino and M. G. Scutella, "Shortest path algorithms in transportation models: classical and innovative aspects," in *Equilibrium and advanced transportation modelling*, pp. 245–281, Springer, 1998.

[87] A. Buluç, J. R. Gilbert, and C. Budak, "Solving path problems on the GPU," *Parallel Computing*, vol. 36, no. 5, pp. 241–253, 2010.

[88] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, and D. Lavenier, "Efficient Multi-GPU Computation of All-Pairs Shortest Paths," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 360–369, May 2014.

[89] K. Matsumoto, N. Nakasato, and S. Sedukhin, "Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pp. 145–152, Sept 2011.

[90] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. ACM*, vol. 24, pp. 1–13, Jan. 1977.

[91] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*, vol. 22. SIAM, 2011.

[92] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, (New York, NY, USA), pp. 177–187, ACM, 2005.

[93] M. Besta and T. Hoefler, "Slim Fly: A Cost Effective Low-diameter Network Topology," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 348–359, IEEE Press, 2014.

[94] R. R. Veloso, L. Cerf, W. M. Junior, and M. J. Zaki, "Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach.," in *EDBT*, pp. 511–522, 2014.

[95] J. R. Gilbert and J. W. Liu, "Elimination structures for unsymmetric sparse LU factors," *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 2, pp. 334–352, 1993.

[96] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, "Parallel FPGA-based All-pairs Shortest-paths in a Directed Graph," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, (Washington, DC, USA), pp. 112–112, IEEE Computer Society, 2006.

[97] G. J. Katz and J. T. Kider, Jr, "All-pairs Shortest-paths for Large Graphs on the GPU," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, (Aire-la-Ville, Switzerland, Switzerland), pp. 47–55, Eurographics Association, 2008.

[98] E. Solomonik, A. Buluc, and J. Demmel, "Minimizing communication in all-pairs shortest paths," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 548–559, IEEE, 2013.

[99] T. Okuyama, F. Ino, and K. Hagihara, "A task parallel algorithm for finding all-pairs shortest paths using the gpu," *International Journal of High Performance Computing and Networking*, vol. 7, no. 2, pp. 87–98, 2012.

[100] Z. Fu, M. Personick, and B. Thompson, "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs," in *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pp. 1–6, ACM, 2014.

[101] D. Gregor and A. Lumsdaine, "The Parallel BGL: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[102] P. Erdós and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hungar. Acad. Sci*, vol. 5, pp. 17–61, 1960.

[103] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." `http://snap.stanford.edu/data`, June 2014.

[104] A. E. Sariyuce, E. Saule, K. Kaya, and U. V. Catalyurek, "Regularizing Graph Centrality Computations," *Journal of Parallel and Distributed Computing*, vol. 76, no. 0, pp. 106 – 119, 2015. Special Issue on Architecture and Algorithms for Irregular Applications.

[105] D. A. Bader and K. Madduri, "SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–12, IEEE, 2008.

[106] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, IEEE, 2004.

[107] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pp. 365–376, IEEE, 2011.

[108] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of gpu kernels using performance counters," in *Green Computing Conference, 2010 International*, pp. 115–122, IEEE, 2010.

[109] S. Hong and H. Kim, "An integrated gpu power and performance model," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 280–289, ACM, 2010.

[110] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–14, March 2007.

[111] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.

[112] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 349–362, ACM, 2012.

[113] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, 2011.

[114] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," 2013.

[115] S. Che, "Gascl: A vertex-centric graph model for gpus," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[116] S. Dalton, N. Bell, and L. Olson, "Optimizing sparse matrix-matrix multiplication for the gpu," 2013.

[117] D. G. Merrill, III, *Allocation-oriented Algorithm Design with Application to Gpu Computing*. PhD thesis, Charlottesville, VA, USA, 2011. AAI3501820.

[118] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, p. 2, 2007.

[119] H. W. Cain, M. H. Lipasti, and R. Nair, "Constraint Graph Analysis of Multithreaded Programs," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[120] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer*, vol. 29, pp. 66–76, Dec. 1996.

[121] S. Qadeer, "Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model Checking," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 14, no. 8, pp. 730–741, 2003.

[122] N. Chong and S. Ishtiaq, "Reasoning About the ARM Weakly Consistent Memory Model," in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC): Held in Conjunction with (ASPLOS)*, 2008.

[123] Y. Chen, Y. Lv, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan, "Fast Complete Memory Consistency Verification," in *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[124] C. Manovit and S. Hangal, "Efficient Algorithms for Verifying Memory Consistency," in *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005.

[125] A. Arvind and J.-W. Maessen, "Memory Model = Instruction Reordering + Store Atomicity," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[126] C. Manovit and S. Hangal, "Completely Verifying Memory Consistency of Test Program Executions," in *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 166–175, IEEE, 2006.

[127] C. Manovit, *Testing Memory Consistency of Shared-memory Multiprocessors*. PhD thesis, Stanford, CA, USA, 2006.

[128] A. E. Condon, M. D. Hill, M. Plakal, and D. J. Sorin, "Using Lamport Clocks to Reason About Relaxed Memory Models," in *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 270–278, 1999.

[129] A. Landin, E. Hagersten, and S. Haridi, "Race-free Interconnection Networks and Multiprocessor Consistency," in *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA)*, pp. 106–115, 1991.

[130] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," in *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, pp. 56–66, 1988.

[131] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[132] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *et al.*, *Introduction to Algorithms*, vol. 2. MIT press Cambridge, 2001.

[133] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang, "Fast and Generalized Polynomial Time Memory Consistency Verification," in *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, Springer-Verlag, 2006.